1997

An object-oriented infrastructure for the development of a computer modeling system for electrochemical analysis and visualization.

Eric. Marcuzzi
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

An Object Oriented Infrastructure for the Development of a Computer Modeling System for Electrochemical Analysis and Visualization

by

Eric Marcuzzi

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor
Windsor, Ontario, Canada
1997

ABSTRACT

In this thesis a partial system design and implementation for a computer modelling system, the **VPMS** (Virtual Prototyping, Modelling and Simulation) system, which supports extensibility, reusability, distributivity and openness, is presented. This system identifies and compartmentalizes modelling system functionality, such as numerical modelling and scientific visualization, giving rise to three class libraries which embody the functionality through layers of software abstraction.

The system is presented in the context of an electrochemical application, although the design is extensible to other application domains. As a partial design and implementation it does not contain all functionality required of a modelling system, however, the design is such that new functionality may be added in a straightforward manner through the addition of layers of software abstraction or functional components (environments).

Dedicated to Rosanna Vitale: Cara, sempre nel mio cuore!

ACKNOWLEDGMENTS

I express my most sincere gratitude and utmost respect to Dr. Robert D. Kent whose guidance, support, dedication and supreme patience have enabled this work to come to fruition. I extend my appreciation to Dr. Mordechay Schlesinger for his direction, support and insight throughout this work (I shall always remember the preacher). I also convey my thanks to Dr. Peter Tsin for his invaluable service on my committee and instruction throughout my tenure at this university. I am fortunate to have been the beneficiary of the knowledge and wisdom of these three gentlemen. Thanks also go to the faculty, staff and graduate students in the School of Computer Science. Honorable mention and many thanks go to Zina Janabi, Robert Yang and Doug Thistle.

I wish to thank my family, most notably my parents, whose generosity and support have known no limit. They have always been present when needed and close by when not (Your timely little gifts have been greatly appreciated). To the congregation of the CFDI, whose presence has been both inspirational and, at times, intoxicating, salute! My warmest appreciation goes to Paolo and Nocciolina who are a constant source of comfort and delight.

Most importantly, thanks go to Rosanna Vitale. Cara Rosanna, non ho le parole per esprimere la mia gratitudine per la tua pazienza, l'appoggio, la forza, e i sacrifici che sono stati la fonte della mia perseveranza. L'espressione di questo riguardo te la offre il mio cuore mediante il mio amore per te.

Table of Contents

LIST OF FIGURES

Chapter 1 INTRODUCTION

1.1 Thesis Goal

The goal of this thesis is to provide a conceptual design and partial implementation of one portion of a computer modelling system, named **VPMS** (or Virtual Prototyping, Modelling and Simulation) for use by research scientists and engineers. The design, which has its major components designed as individual self-contained environments, is to support the creation of a loosely coupled, extensible, modular, open, distributed system capable of accessing a large number of visualizations. Furthermore the design is to provide a structure which supports continued incorporation of modelling functionality, not one which provides all required functionality.

The design is achieved through the identification and compartmentalization of modelling system functionality. This functionality is implemented through layers of software abstraction which is contained within several class libraries forming the basis for the contribution of this thesis to computer modelling.

1.1.1 Context

The completeness of the design is within the context and scope of the electrochemical application discussed in this chapter and throughout the paper. The system satisfies a set of general requirements (e.g. modular construction) as well as those which are specific to the electrochemical context. The functionality and design of the **VPMS** system is a subset of the larger domain of modelling.

The system should not be limited to this single domain but be extensible to other research areas through addition, or simple replacement, of appropriate environments, smaller functional components (or modules) and layers of software abstraction .

Based on an incomplete specification of requirements provided at the onset of this work, an early system design was conceived and a prototype constructed. The resultant design and implementation was evaluated and subjected to a redesign, based on additional information and set of requirements. This development conforms to the prototype philosophy of software development [Pressman 92] consisting of a design, implementation, evaluation and redesign life cycle.

The system described in this thesis is the product of the redefinition of the earlier system design. The implementation issues discussed are also in the context of the revised system design. It should be noted that much of the earlier implementation is viable for use within the new revised system implementation. Its inclusion requires extensions, modifications and reorganization into new (objected oriented) class structures to support the new design. These modifications are discussed in Chapter 5.

The implementation does not represent the entirety of the design but embodies the initial stages of the development of the structure and software layers.

The aggregation of associated issues and research areas (discussed below) and the amount of complex functionality which a computer modelling system offers makes the task of engineering such a system large and complex. Modern software engineering provides methodologies for managing such a project. Software engineering principles provide structure for problems associated with the gathering and specification of user

requirements, the generation of general and detail system designs, the coding and maintaining of a software projects on a large scale.

1.2 Modelling

The field of modelling covers a vast scope with respect to research areas which must be combined to serve modelling endeavors. Two of the most significant research areas are those concerning numerical modelling and scientific visualization. It is required that the appropriate function, position and interaction of each within a modelling system be delineated. Other issues include:

- Class of models used within the area of modelling. i.e. structural numerical, behavioral.
- Behavioral theories and governing laws which drive the models.
- Application area. i.e. Two areas within the scientific domain are biological modelling and chemical modelling.
- Intended user group and nature of required interaction as dictated by the application domain.
- Nature of output of a model.
- Design of models and the system in which they work.

The convergence of these issues is necessary for the development of high-level tools for modelling which are relevant and useful in areas of research and development. As stated in [Haber 90] the specification of high-level tools in addition to a high-level system integration are required to support the more technical and application specific requirements of a modelling system.

The remainder of this chapter serves to introduce scientific and numerical modelling, scientific visualization and specific application domain.

1.2.1 Numerical Modelling, Visualization and Electrochemistry

The term model (e.g. numerical model) refers to a representation of a real entity, process or concept while the term modelling is the process of creating and/or using a model. The purpose of scientific modelling is to study behavioral characteristics of a real entity or process.

For any given element of reality, there may be numerous representations, or models, which can be created. Different types of models include clay, pictorial, mechanical or mathematical, to name but a few. Different types of models are used for different reasons and the choice of model depends on the requirements placed upon it by the researcher. It is important , therefore, that an appropriate model be selected. Consider. For example, an engineer investigating new gearing mechanisms might choose different types of models to aid development, but for purposes of experimentation all but the mechanical or mathematical models would be insufficient.

The value of models lies in the opportunities they provide users to observe, experiment with, and understand what the model represents without having to commit to potentially large production costs or time scales. The value is also apparent where it may be too dangerous to interact with the real entity or physically impossible. For instance, it is impossible to tilt Jupiter about its axis by 20 degrees to determine the effects.

Although there are many different types of models available, it may be difficult to acquire one which is cost effective, timely and adequate for research and engineering purposes. Sub-atomic particle research may have some use for pictorial renderings of particle motions but these models do not possess detailed descriptions of particle

behavior required to truly represent a sub-atomic system. A more appropriate model which can serve as a true representation, is one which is subject to the same governing laws and behavioral interactions that the real entity or process is subject to. Mathematical models can often meet these requirements.

Mathematical models provide users with accurate and powerful representations of reality because they are created using the same mathematical relations and functions that govern the operation and existence of a real entity or process. Due to computational limitations it is only possible to create a usable model to a certain maximum degree of precision or accuracy.

The nature of mathematical models is such that large and complex systems of equations must be solved at hundreds of thousands (possibly millions) of sample points. Because of this complexity it is common practice for these types of models to be implemented using a computer. To perform these calculations on a computer, computationally complex numerical methods are used thus demanding a large portion of CPU time in a modelling application.

The output from mathematical models is also mathematical in nature, represented as equations, functions or raw numerical data which is often difficult to interpret, understand and use.

Computer graphics is a research area concerned with methodologies for generating graphical images on a computer display. Computer images are visual representations of one or more sets of data. The relationship between computer graphics and mathematical modelling (or simply modelling) can clearly be seen; models produce

numerical data and computer graphics consumes numerical data to produce visualizations which aid in interpreting and understanding the data. The term scientific visualization is commonly used to describe the use of computer graphics to create visual images relating to scientific domains. This term will be used throughout this paper for the same purpose.

The combination of these two fields is known as computer modelling, or alternatively, computer simulation. Henceforth, the term computer modelling will be used to describe the research area. Computer modelling (system) or simply modelling (system) will be used to describe systems which support computer modelling.

A particular area of interest in electrochemistry is electrostatics. Research is concerned with issues relating to the properties and behaviors of static electric fields about a sample element geometry; henceforth referred to as a plate geometry. Sample geometries contain various charged, uncharged and neutral components which can be composed of a number of possible materials and placed in varying configurations and relative positions. Specific interest lies in the effects on the electric potential and electric fields as different geometric, spatial and physical attributes are varied.

The computation of the electric potential $U(x,y)$ requires the solution of the Laplace/Poisson equation,

$$\nabla^2 U(x,y) = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) U(x,y) = b(x,y) \text{ [Davis 84]}$$

at various points on the plate geometry defined by a rectangular grid, or mesh, which overlays the geometry as seen in the Figure 1.1 below.

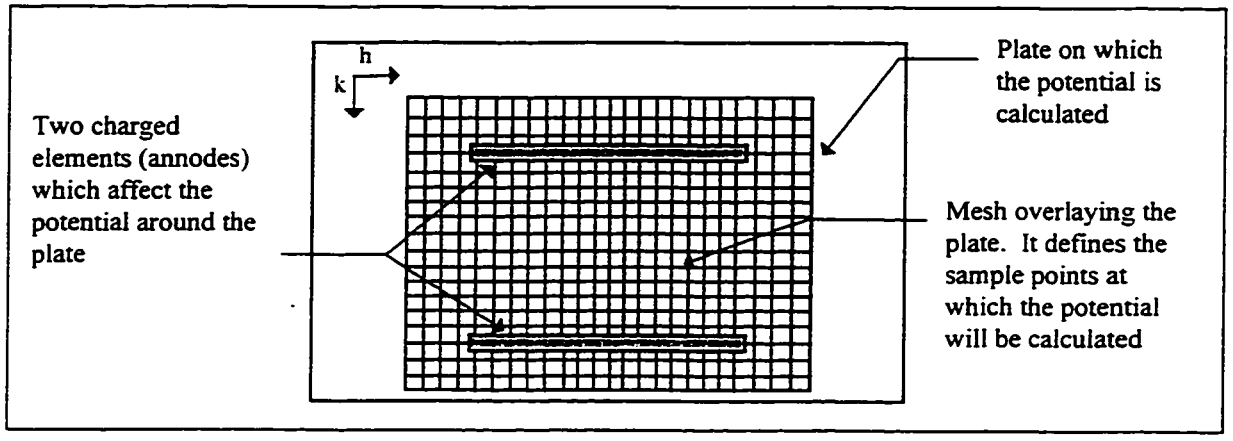

Figure 1.1: An example plate geometry overlaid with a mesh. The electric potential is calculated at each grid point. h and k represent step sizes of the grid in the x and y directions respectively.

The potential value at each of these sample points can be calculated using Taylor's expansion for $\nabla^2 U(x,y)$ and the points surrounding a particular sample point of interest. For example, given a point (x_i,y_j), and step size h along the x axis (see Figure 1.2), the potential value $U_{i,j} = U(x_i,y_j)$, can be expressed as,

$$\frac{\partial}{\partial x} U_{i,j} \approx \frac{U_{i,j+1} - U_{i,j-1}}{2h}$$

$$\frac{\partial^2}{\partial x^2} U_{i,j} \approx \frac{U_{i,j+1} + U_{i,j-1} - 2U_{i,j}}{h^2}$$ [Davis 84]

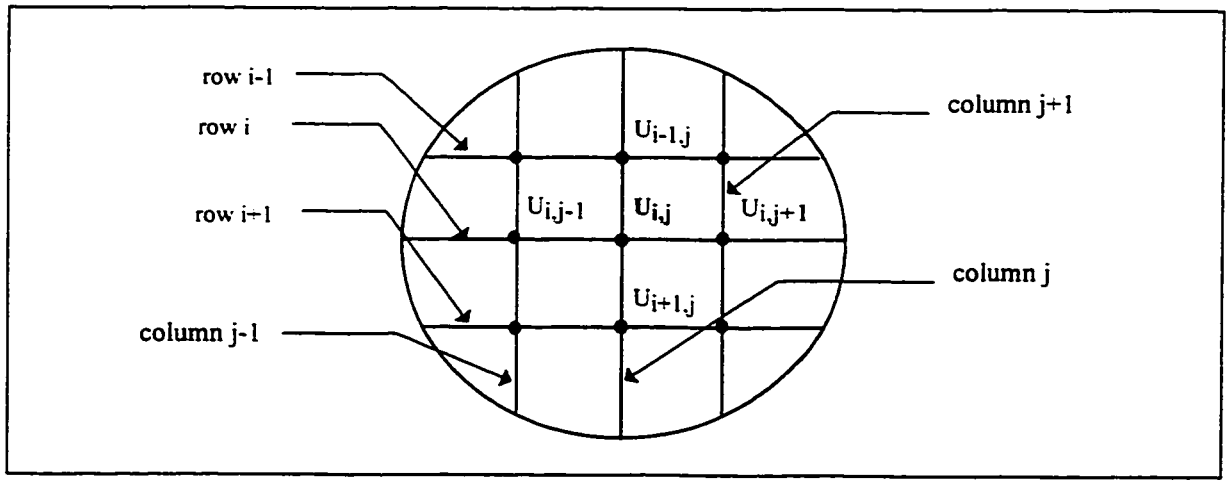

Figure 1.2: This diagram represents a magnified view of Figure 1.1 illustrating the sample point $U_{i,j}$ and the surrounding points.

This software system provides the capability for interactive specification of input geometries, selection and activation of numerical models and access to suitable visualizations for interpretation of data. Figure 1.3 illustrates the aggregation of research areas and issues into a coherent whole which contains appropriate modelling system functionality.

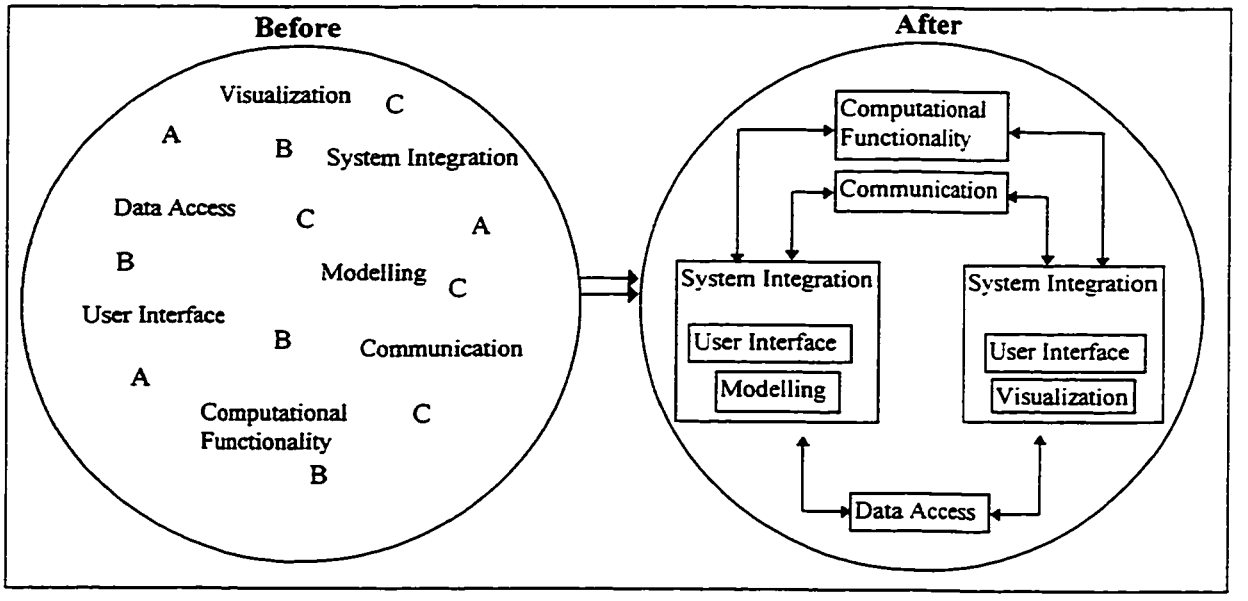

Figure. 1.3: Illustration of the transformation from a disorganized collection of information to an organized, structured framework which encompasses modelling system functionality.

1.3 Thesis Layout

The second chapter provides necessary background information regarding models, modelling and scientific visualization (component research areas) which are necessary for a computer modelling system.

The third and fourth chapters discuss the design and implementation issues respectively. Throughout these two chapters elements of the list above are considered in the design and implementation of a computer modelling system.

Chapter 2 BACKGROUND RESEARCH

2.1 Introduction

It is the system in which a model operates (the modelling system) which provides the interactive functionality to facilitate research and development. Developing a modelling system requires an effective organization and communication of modelling and visualization components and this organization aids in achieving the necessary functional and non-functional system requirements.

Due to the expansive nature of computer modelling a general understanding of modelling and visualization is required to identify the proper context for the design and implementation of a computer modelling system. Examining current paradigms and systems demonstrates appropriate design, interaction, abstraction and implementation considerations which are addressed in the creation of a system for various applications and end-users.

In addition to current modelling systems, an examination of visualization systems is also required because of its importance to computer modelling. An examination of both practical as well as theoretical systems will aid in the identification and specification of directions, goals and methods of implementation of visualization used in conjunction with modelling.

In this chapter we discuss modelling and current modelling systems first and then proceed to examine the field of scientific visualization.

2.2 Modelling

This section will focus on two types of systems that are relevant to the discussion of modelling. The first type is a user oriented system and is represented by a product known as **Maxwell™**. User oriented systems will refer to systems that are directed towards users who are not familiar with computer programming. The second type of system is a programmer oriented system represented by a product such as **Maple™**.

2.2.1 Maxwell

Maxwell™ is representative of systems which contain a previously constructed mathematical model and allow the user to specify parameters to this model in some convenient manner. The model then proceeds to simulate some phenomenon through mathematical calculations and produce data. The user may then visualize or perform analyses on the data using post-processing capabilities provided by the system. It is a modelling system directed more towards end-users rather then creators, developers or system managers who require knowledge of its internal data structures and operations.

Maxwell™ is a product designed specifically for electromagnetic field simulation. The system allows the user to graphically construct representations of actual physical components which a model will use in its simulation as well as designate environmental and individual material properties. In the case of the Electrostatics model, a plate geometry is constructed from a group of polygonal objects which represent a configuration of actual conductive and non-conductive elements such as plates, substrates, electrodes, anodes and cathodes. The model calculates the electric potential

about the geometry based on the configuration of the elements as well as their individual properties. A plate geometry is illustrated in Figure 2.1 below.

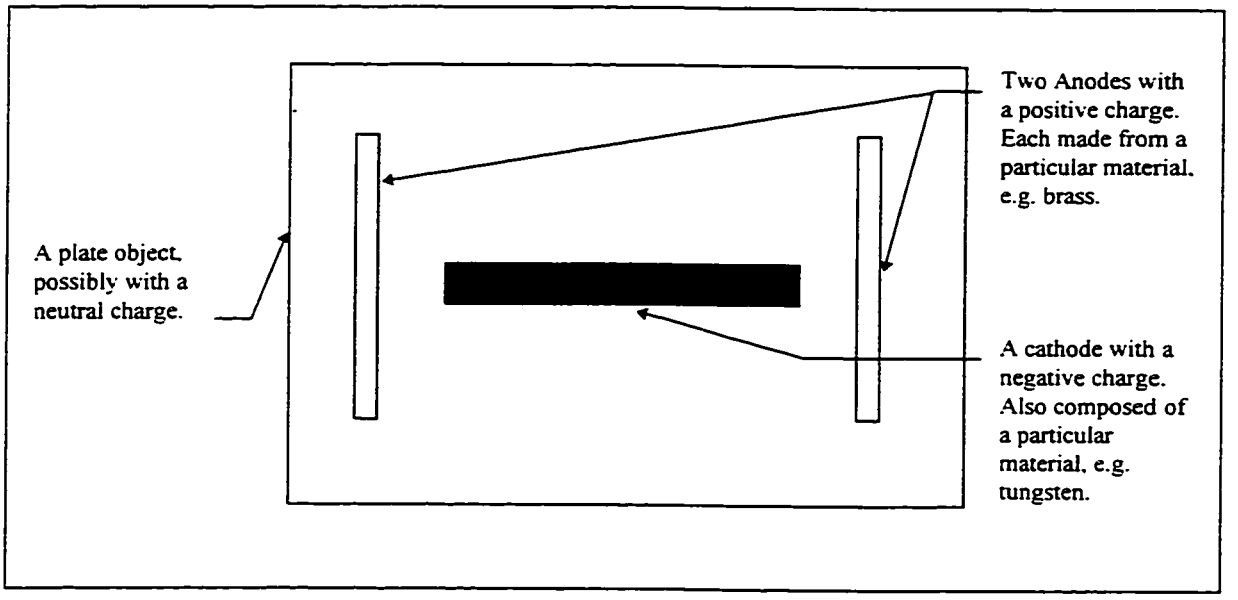

Figure 2.1: Each element on the plate geometry possesses specific properties which are used in the calculation of solutions.

Maxwell™ provides a post-processor for visualizing computed quantities. Using the post-processor, quantities such as electric field vectors, **E** and **D**, and the capacitance, C, can be computed and visualized. A user may also manipulate quantities from within the post-processor and visually examine the effects. This gives the user the ability to better understand the behavior of the system based on results from simulations.

The visualization capabilities of the post-processor in **Maxwell™** , for the electrostatics model, consists of contour plots, color/shaded plots and arrow plots of various fields of values (both scalar and vector fields are represented). Some of the functionality which the post-processor offers includes:

- The display of visualizations; either individually or together (overlapping one another).

- The display of the sample mesh in either shaded or wire frame representations.
- The magnification of areas of the visualization output.
- The superimposition of axes and grids on visualizations.
- The modification of geometry object attributes, such as color.

Other post-processing visualizations, like the generation of cutting planes, are available with different models, such as the Magnetic Force model. The following diagram is a reproduction of a **Maxwell™** post-processing visualization.

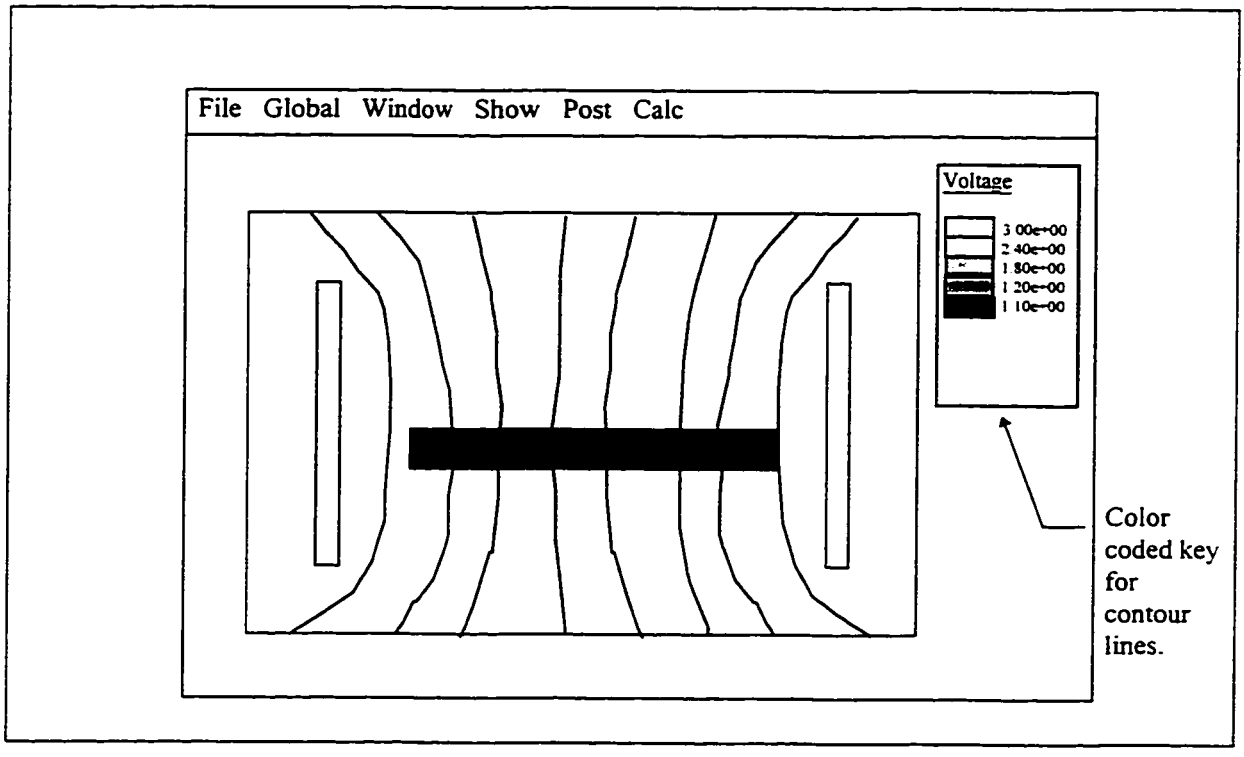

Figure 2.2: This is a representation of a contour plot of voltage values produced by the **Maxwell™** post-processing facilities.

The operation of **Maxwell™** takes place through the executive commands window illustrated in Figure 2.3. This window guides the user through the process of selecting the model (electrostatic, magnetostatic, etc.), drawing the plate configuration, assigning the desired element materials and boundary conditions, solution parameter

specification (mesh refinement, error approximation, etc.), and post-processing. Each step provides the user with a different user interface window from which the appropriate parameter entries can be made or visualizations can be viewed. The method of operation leads users, step by step, through the setup and execution of a model and clarifying the necessary steps and reducing setup errors, however, it limits them to a specific sequence of choices (must perform step c then step d) leaving little freedom for alternative scenarios for system usage.

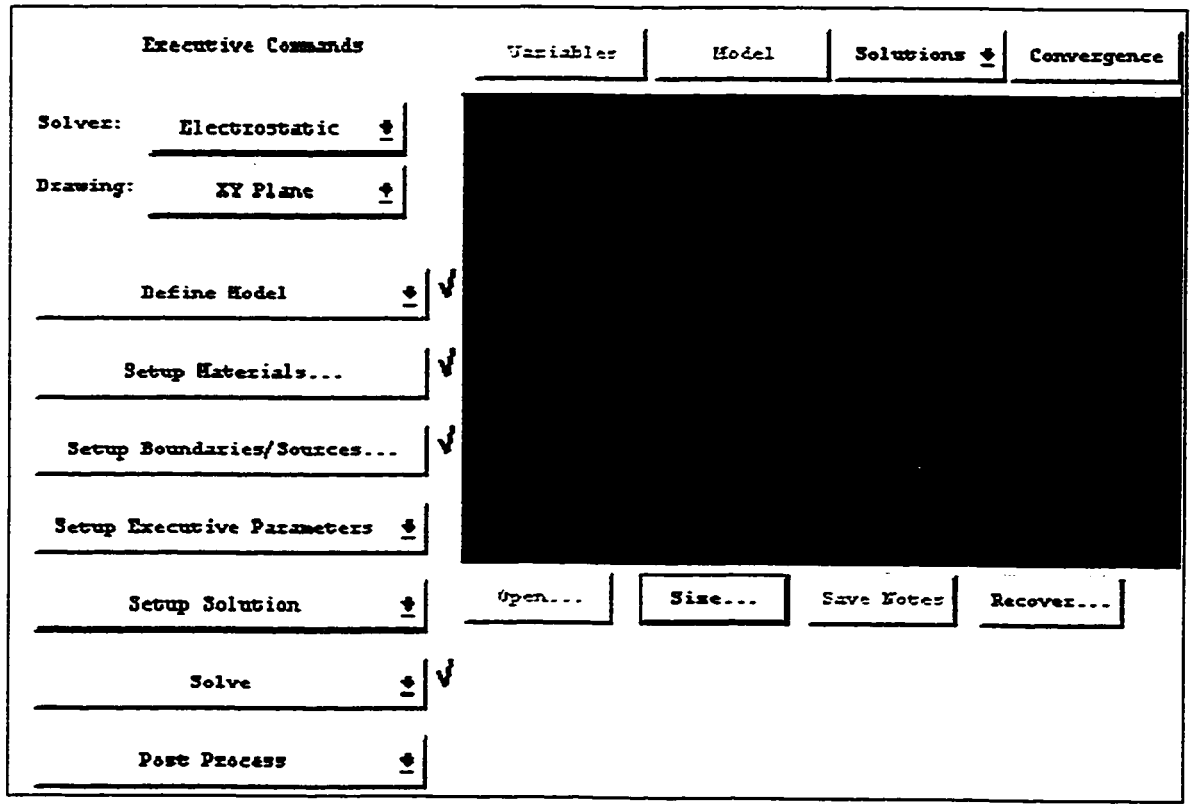

Figure 2.3: The executive commands window for Maxwell™. It is the main interface window from which the user access the systems functionality.

Maxwell™ provides a relevant set of modelling system functionality which is easily accessible through a structured interface making it a desirable system for end-users. **VPMS** will provide a similar style of structured interface to access functionality.

Although **Maxwell™** provides effective organization and access to modelling system functionality, it is limited. There is no support for distribution of modelling system functionality, the functionality which is available is narrow in scope and there is no support for extension of the system except for its creators; it is a closed system.

The limitations of **Maxwell™**, described above, illustrate goals of the **VPMS** system which may be achieved through a different design.

2.2.2 Maple

Maple™ is an algebraic and symbolic manipulation system. It allows a user to create, interact with, test, modify and prototype mathematical models which may be incorporated into other computer modelling systems. It is a system for programmers of numerical models.

The **Maple™** system provides a vast library of functionally high-level mathematical routines, defining functionality for Calculus, Linear Algebra, Euclidean Geometry and Combinatorial Mathematics, to name but a few, in addition to an environment in which they can be used. These routines are available to a user through the **Maple™** command line input prompt or through the **Maple™** programming language. Users take advantage of the power of the **Maple™** system by creating programs as a combination of many **Maple™** library routines and user created functions and procedures. Together these procedures and functions, along with standard language constructs and functionality such as those for decision, control and access to low level system functionality, perform specific user defined tasks, such as a numerical simulation.

Using **Maple™** as an environment for the creation of a numerical model requires the user to have computing and computer programming knowledge because they must be able to generate a program flow, or specification, create and use appropriate functions and procedures as well as required data structures. The following figure contains a portion of **Maple™** code which is part of a program to generate a bezier curve given a list of control points. The Figure illustrates the nature of the use of **Maple™**.

```
> MyBezier := proc(list_of_points)
>    if not(type(args[1],list)) or not(type(op(1,list_of_points),list)) then #check for improper input
>       ERROR(`Wrong type argument to`, procname, `:`, `list of lists is expected`);
>    fi;
>    plotinfo := [];#initialize to empty list
>    L := nops(list_of_points)-1;#L is the number of control points -1, i.e. there are L+1 ctrl points
>    for t from 0 by .01 to 1 #set the range of t, there will be 100 samples btwn each of the ctrl pts
>    do
>       for k from 0 by 1 to L
>       do
>          #calculate the Bernstien poly and multiply it by x and y components of each point
>          interimX[k] := op(1,op(k+1,list_of_points)) * Bernstien(L,k,t):
>          interimY[k] := op(2,op(k+1,list_of_points)) * Bernstien(L,k,t):
>       od;
>       totalX := sum(interimX[i],i=0..L):#for each value of t The Bernstien poly's are multiplied by the
>       totalY := sum(interimY[j],j=0..L):#the points P0 to PL and summed
>       plotinfo := [op(plotinfo),totalX,totalY];#list containing a sequence of all new x and y values for
>    od:                                          #the bezier curve
>    plot1:=plot(plotinfo,style=LINE,color=RED):#plot the bezier curve
>    plot2:=plot(list_of_points,style=LINE,color=WHITE):#plot the original ctrl points
>    display([plot1,plot2],title=`Plot of Control Points(white) and Bezier Curve(red)`):
> end:
```

Figure 2.4: This code (procedure) is part of a **Maple™** program which calculates a Bezier curve given a set of control points. The ">" characters represent **Maple™** input prompts.

Maple™ provides an extensive array of plot capabilities both in 2 and 3 dimensions (henceforth referred to as 2D or 3D). These plots represent the output from the computation of mathematical functions and are generally plotted on a set of axes in either 2D or 3D space. Visualizations of this nature can often be difficult to interpret and are generally only useful to mathematicians. The plotting options which **Maple™** provides are quite extensive and aid in highlighting details in the visualizations. Some of

the plotting options include viewing perspective manipulation, plots in alternative coordinate systems, different types of axes, multiple rendering styles such as wireframe and patch, as well as different lighting and shading options. These options are accessible at a high-level of abstraction, through interface mechanisms (e.g. menus), or at a low level, through library functions and data structures. An example of the low level use of the plotting facilities can be seen in the last four lines of Figure 2.4. Figure 2.5 illustrates a straightforward 2D **Maple™** plot.

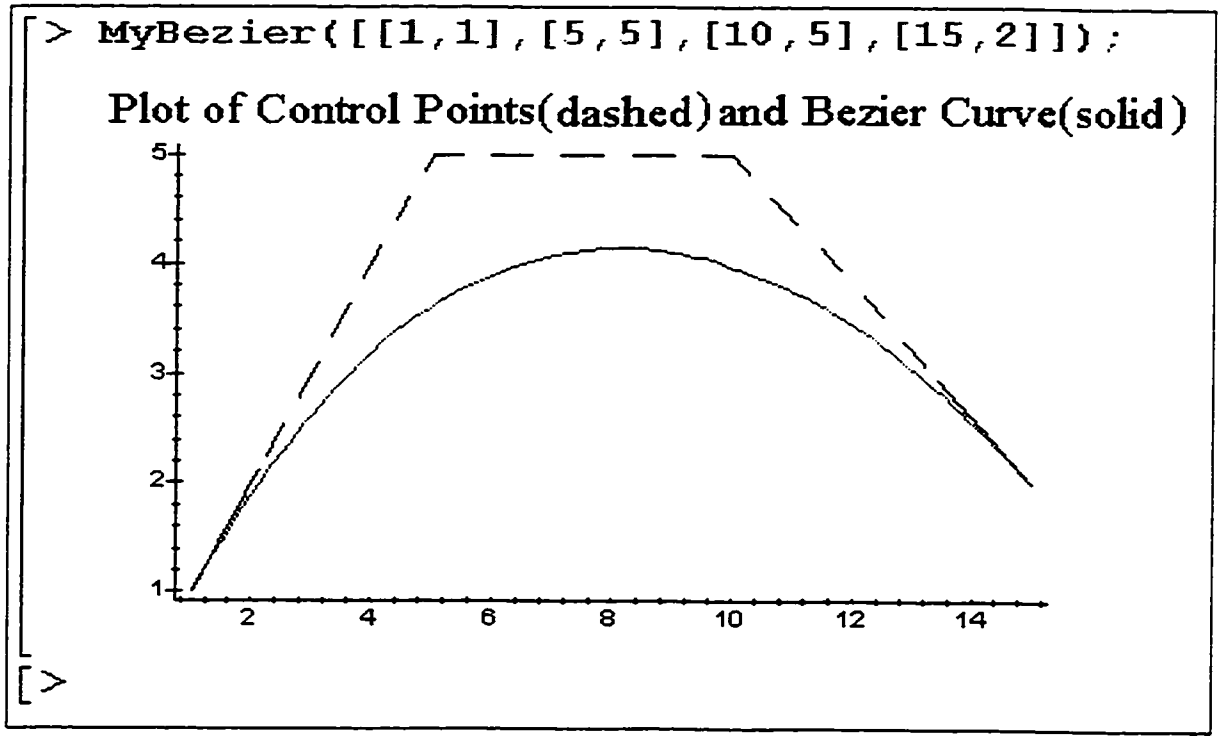

Figure 2.5: This diagram illustrates the plot of the program of Figure 2.4. The plot command is embedded in the program "MyBezier()". The plot command creates data structures which are used to display the image. In Figure 2.4 two plot data structures, "plot1" and plot2", are created and displayed using the display(). Plot commands may be entered via the command line as well.

Maple™ allows users to program their own modelling functionality due to the availability of appropriate abstractions (**Maple™** library functions). This ability is

required of an open system and is not available in a system such as **Maxwell™** which provides pre-defined modelling functionality. Programs created in **Maple™** can be exported as **C**, or **Fortran** code. This positive attribute demonstrates that, in addition to providing appropriate abstractions supporting the creation of models, the **VPMS** system should also provide abstractions which allow the incorporation of numerical models such as those created using **Maple™**.

As an end-user oriented system, **Maple™** lacks an appropriate user interface. Once a model has been created, the only manner in which input can be provided is through command line text based methods which must be explicitly programmed. **VPMS**, as stated previously, provides a more flexible method of input similar to a system like **Maxwell™**.

2.3 Visualization

Visualization for purposes of interpreting data has become a necessary element of computer modelling systems. In this section we discuss scientific visualization as well as several visualization systems and their philosophy of construction.

2.3.1 Reference Models

A visualization system used in any scientific field is required to be able to:

- Handle large data sets (typically 10's of MB for electrochemical analysis applications).
- Provide a number of useful visualizations.
- Be as simple for the user to operate as possible.
- Efficiently perform numerous computational tasks including scene rendering, interpolation and data mapping.

A visualization system must satisfy these requirements and do so within diverse fields having differing data requirements with respect to structure, format, dimensionality and type.

Different data structures represent the data in different physical layouts, such as arrays of different sizes and dimensions. Different data formats may include any compression or ordering of information within a data file. Dimensionality of data includes scalars, vectors, tensors and other multi-dimensional data which a visualization depends upon. Types of data include integer, floating point, vectors, n-tuples, ordinal data, dependent variables versus independent variables, etc. The issues of data representation, together with the status of visualization as a maturing field and the lack of coordination of system development are contributing factors to the abscence of a reference model for scientific visualization.

The existence of one or more reference models for visualization would serve designers of systems by supplying a standard specification for creating systems which are extensible, reusable, efficient and useful across many application domains. Reference models should provide a non-application specific abstract framework which the creator of a system would then use and modify to suit application specific needs. The use of such a reference model would result in more rapid development of systems which have a wider application base. The discussion of models not only applies to systems but to data as well. Standardized data models would increase the portability of data from one system to another [Butler 93].

The lack of a reference model illustrates the non-trivial requirements of visualization and the difficulties inherent in creating visualization systems as well as incorporating visualization as a fundamental requirement and component of a computer modelling system. A reference model provides insight into requirements and possible design for a modelling system, but the absence of such models should not prevent systems from being developed independently.

2.3.2 Current Systems

The many systems which have been previously developed can be partitioned into two broad categories based on their underlying paradigm of construction and operation. The first category encompasses those visualization systems which follow the Dataflow paradigm and the second consists of all of the others. The systems in the second group are referred to as "turnkey". [Shroeder 92] describes these as systems which are designed for an end-user because no programming knowledge is required.

2.3.2.1 Dataflow

The Dataflow architecture for scientific visualization systems was introduced in 1989 in the form of a visualization system known as AVS, or the Application Visualization System [Ribarsky 92] [Upson 89] and since that time many new systems have embraced this paradigm. The paradigm centers around the concept of a sequence of modules which must be coupled together in a particular configurations to achieve a desired visualization.

The name Dataflow reflects the method of operation of the paradigm. Data to be visualized flows through a sequence of modules, each one performing some form of data transformation, mapping or other manipulation with the output from one becoming the input to another. The result of the execution of this aggregate of modules is a completed visualization. In this respect the paradigm is similar to a functional programming paradigm where each module represents a function and the system executes as a composition of functions to produce a final output.

Data transformation, or mapping, is central to the Dataflow concept. Raw data input into the system is generally unsuitable for immediate graphical display. Often, some form of interpolation or filtering must be performed. For example, the output of an electrostatic simulation is a set of potential values at specific points on a plate geometry. To produce an arrow plot visualization of an electric field an interpolation is required to determine potential values at other points in the interior of the geometry. Using the Dataflow concept the original potential values would be given to an interpolation module whose output would be additional, interpolated, potential values. These values would then be used to calculate the electric field, **E**, across the plate. The calculation of **E** would require one or more modules which receive the potential values as initial input and produce electric field values as final output. The final stage involves the mapping of data values to graphical primitives and attributes and the mapping of graphical primitives to a display.

Generally, a visual programming interface is provided for the utilization of a Dataflow visualization system. To create a visualization using this type of interface the

user must construct a network Dataflow graph. This is achieved by graphically connecting together boxes (representing modules in the system) in a specific configuration so that the flow of data through the sequence of boxes results in a desired visualization. Some systems transform the graphical network hierarchy created by the user into a script which is interpreted by the system. Alternatively, some systems let the user specify the Dataflow network as a script which is given to the system for interpretation. Figure 2.6 is an illustration of the above arrow plot example.

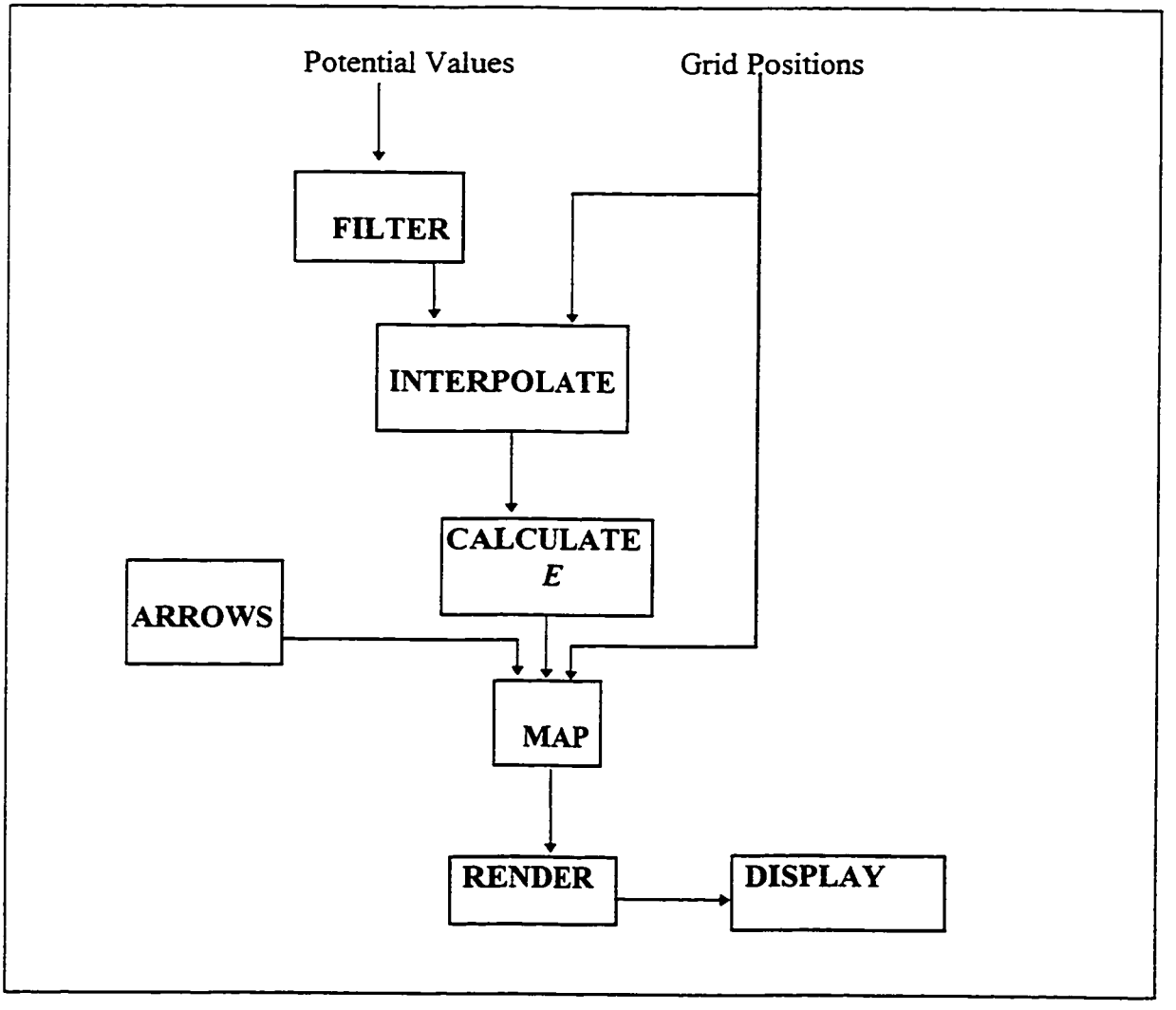

Figure 2.6: This illustrates the network flow graph that a user must design to create a new visualization. This network flow graph is a representation required to generate an arrow plot visualization.

Dataflow systems may offer three types of interaction based on the visualization programming knowledge of the end-user. First, the systems may have one or more network hierarchies which are capable of providing specific visualizations which a novice end-user can employ. Either these hierarchies would be supplied with the system or created by another user. Second, the system allows more experienced users to create their own custom visualizations with existing modules by designing and constructing network flow graphs, as in the diagram above. The third form of interaction allows expert users to program their own modules and include them in the system. These three levels of interaction result in a flexible system for different types of users [Ribarsky 92].

Dataflow systems are modular in design and take advantage of the concepts of the object oriented design paradigm which include module reuse, extensibility and flexibility. Note that an object oriented language need not have been used in the system implementation. A further advantage resulting from the object oriented philosophy is the possible distribution of modules across heterogeneous networked computer systems.

Creating a network flow graph requires the user to select from a large collection of modules and connect them as pictured in Figure 2.6. This mode of interface is often difficult for end-users to work with as they must have some knowledge of the data and its structure. For example, since the input from one module is the output from another, the user must be sure that the type and format of the output data is appropriate for input into another module. If data types do not match then some form of transformation is required, increasing computation time as well as the time and effort needed to create the hierarchy.

A knowledge of the available modules is crucial in the design of a visualization. The user must ensure that the required modules are available, prior to the creating a visualization, so that problems and wasted time are avoided if it is discovered that a needed module does not exist.

Users may create the desired modules themselves but this requires a knowledge of the system, its data structures, data types, the programming language and programming methodologies. This method of creating visualizations is also difficult because it implies that the user must have a knowledge of the types of modules in the system, since they must be integrated with newly created ones, a difficult task for a large number of modules.

Dataflow systems are not task oriented and therefore require end-users to concern themselves with the machinations of creating a visualization. Users of visualization systems do not wish to invest their time constructing them, at any level of abstraction. This time is better spent on activities germane to their area of research and not on the many issues relating to the creation of a specific visualization, i.e. programming languages, data structures, algorithms, etc.

Dataflow systems do not support the concept of the "have data want right picture(s) paradigm" discussed in [Treinish 92]. This refers to the fact that once data is available an end-user wants to immediately select and use an appropriate visualization.

The Dataflow paradigm provides a powerful mechanism for scientific visualization which is extensible, distributable modular and supports code reuse. This is achieved through modularized construction of visualizations.

The Dataflow paradigm illustrates that a more effective user interface is required to access the underlying functionality of a Dataflow system. A system which is task oriented provides specific and relevant tools (visualization modules described above) allowing users to concentrate on using a visualization to achieve their goal without being concerned with its construction thus supporting the "have data want right picture(s)" paradigm.

An additional problem with Dataflow systems is that they are too general and do not provide an appropriate domain specific context required of a visualization system.

2.3.2.2 Non-Dataflow Systems

The other category of visualization systems encompass all those systems which are not Dataflow. Because this category cannot be represented by a single paradigm it is difficult to furnish a comprehensive description of the underlying principles behind the operation of its representative systems. To better understand some of these principles several systems are highlighted and discussed. These systems use different architectures to address the issues of extensibility, modularity, appropriate visualization types, efficiency and distributivity.

The architecture of the **FAST** visualization system [Bancroft 90] emphasizes the performance and handling of the CPU, memory and graphics. In an attempt to achieve their goal the system is implemented as a group of modules which are controlled by a central hub. These modules communicate using shared memory, a form of UNIX interprocess communication, which the hub controls. Modules cooperate to produce

geometric models of data. These models are displayed from a central module which handles rendering. Figure 2.7 illustrates the **FAST** model.

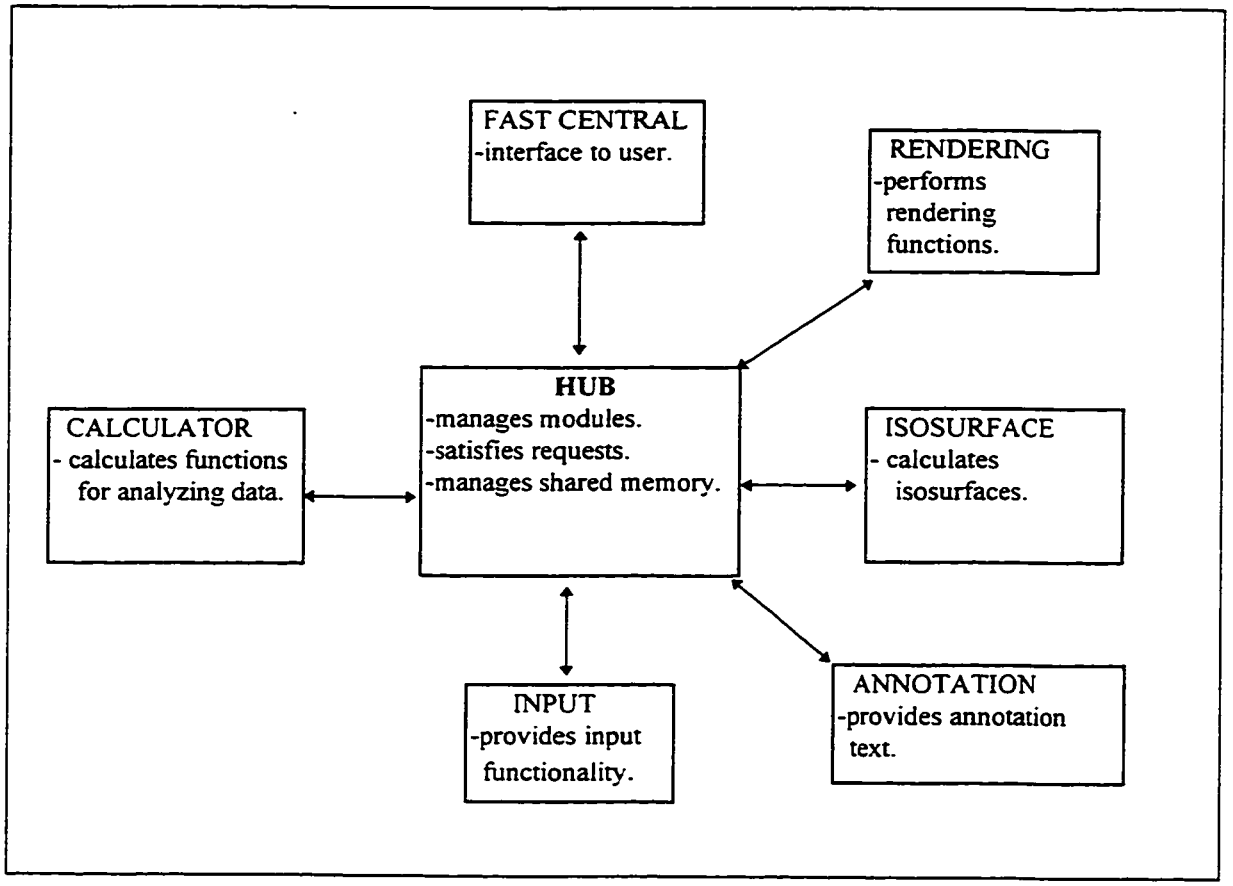

Figure 2.7: The FAST system model.

The hub processes requests from each of the modules such as requests for memory allocation or available data. All memory is managed by the hub and other modules have access to it. Although **FAST** uses shared memory it cannot be distributed across different machines because shared memory across different platforms is not possible. The user access system functionality through a control window ("FAST Central"). From this vantage point the user can invoke all other system functions.

The ability of a visualization system to support multiple visualization types and to be implemented in a distributed and heterogeneous networking environment, with respect to data I/O, are objectives of "tool-based" systems as discussed by [Brittain 90]. These systems are composed of specific tools, i.e. visualization tools, which are generally controlled by one or more managing system layers.

The tool-based system in [Brittain 90] is composed of multiple layers. A component for the management of user interface details is built on top of another layer which manages the various visualizations, i.e. tools. In this case the "Visualization Tool Manager", as it is referred to, manages visualization tools which are implemented as true object oriented objects. Each of these tools can access other system layers for purposes such as data access. New visualization tools may be added to the system without affecting pre-existing ones.

Additional system modules are used for purposes of database I/O. These modules can be distributed among separate machines to provide remote data access. It is therefore possible to use high performance, compute intensive machines for the generation of the data which can then be access by the visualization system which is located elsewhere. Figure 2.8 illustrates the tool-based system of [Brittain 90].

Both of these systems demonstrate alternative methods for providing visualization functionality through different organizations of modules. These organizations support various levels of distributivity, extensibility and modularity.

A problem with the architectures, as a result of the organization, is the dependency of modules, or layers, upon others. Extending, updating or distributing the

system is potentially difficult because of the level of integration of layers and the type of communication between layers and/or modules.

In the case of [Brittain 90] one system layer controls common functionality of a group of modules, for example the "Visualization Tool Manager" controlling the visualization tools. In such a case it is problematic to change the structure or operation of a single tool (without affecting other tools) when the particular functionality which is to be changed is shared among all tools and has been abstracted into a controlling layer. An example is the functionality to render to a display.

VPMS addresses this issue through a reorganization of modules and functionality.

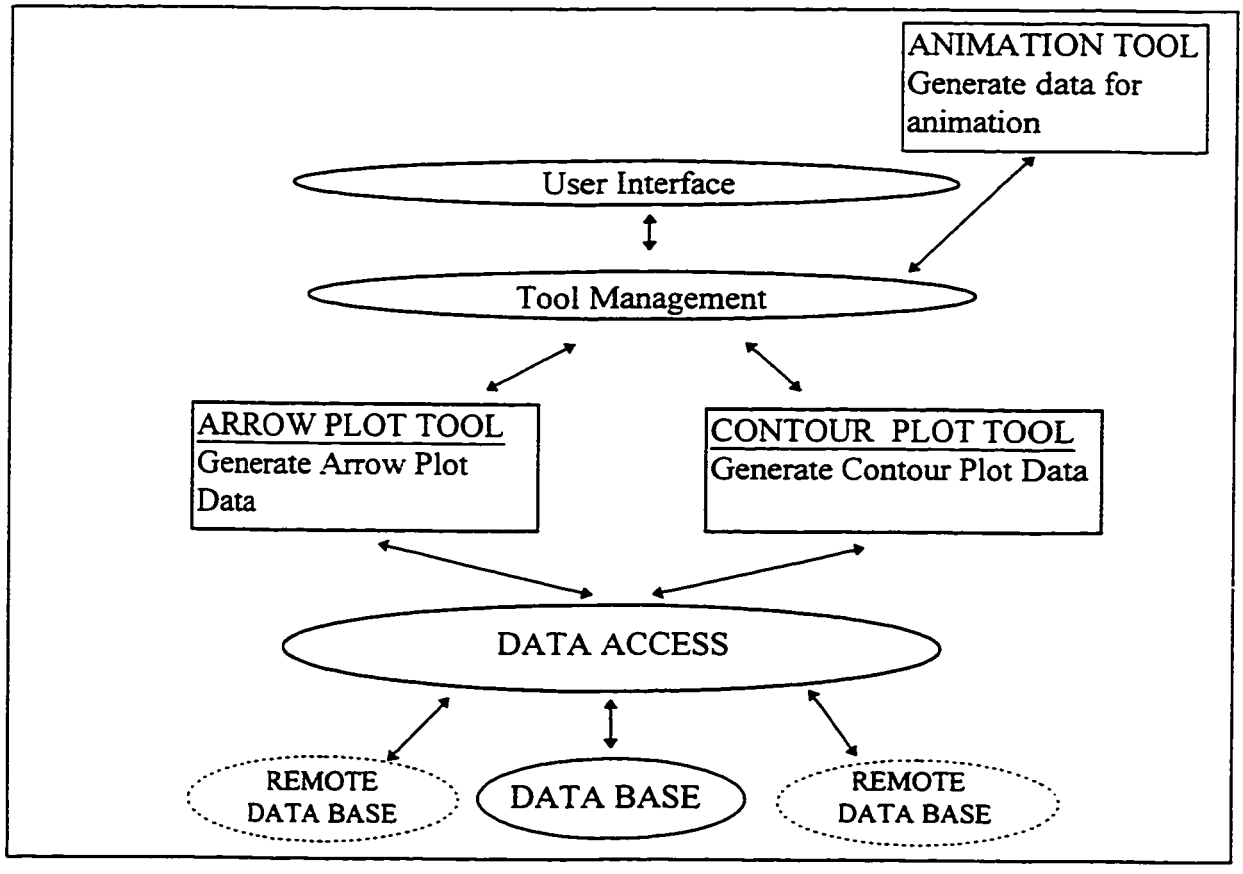

Figure 2.8: A tool-based system model. Ovals represent system layers. Squares represent tools.

To produce systems which emphasize extensibility and flexibility, experiments have concentrated on design and creation of flexible visualization environments. SuperGlue [Hultquist 92] is an environment which promotes the rapid development of robust visualizations for practical use and evaluation as part of the SuperGlue environment. Essentially, it is an easily extensible visualization system, targeted at programmers, for computational fluid dynamics exploration. Extensibility refers to the incorporation of new visualization techniques.

The SuperGlue environment is created as a combination of two languages. **C** is used for computationally intensive routines, such as the numerical and graphical routines, because of its relative speed. An interpreted object oriented language called Scheme is used for control and user interface creation because it is a portable and high-level language.

SuperGlue provides an environment with over 150 classes which are used for creating the visualizations. The strength of SuperGlue is that the programmer can rapidly extend the environment. New visualizations are implemented by creating new classes and methods, improving existing ones or extending the user interface. Classes are quickly created, using the Scheme language, which have access to other objects as well as compiled C routines. New C routines may also be created if necessary.

SuperGlue is a single purpose, "monolithic" system with specific application to computational fluid dynamics. It demonstrates the importance and value of rapidly extensible visualization within a modelling system. Its monolithic nature restricts its development as a more general purpose visualization system which can be distributed and

extended to other application domains because additions are primarily intended to be in the context of computational fluid dynamics.

Similar in nature to SuperGlue is the **Visualization Toolkit** described in [Schroeder 96]. The **Visualization Toolkit** is a C++ class library which supports the creation of visualization applications. This library provides a large set of classes for graphics and visualization. It differs from SuperGlue because the classes do not constitute a visualization system nor are they specific to any particular system. The classes of SuperGlue are not general purpose and are not for use outside of the SuperGlue visualization system.

Like the **Maple™** library the **Visualization Toolkit** library is targeted at programmers who create visualizations of data from the available C++ classes. The **Visualization Toolkit** is a consumer of numerical data and a producer of geometric data which is rendered to a display. It must therefore be "wrapped" with other functionality to be useful as an application. It provides a general purpose layer of abstraction for visualization functionality and, therefore, is not restricted to a specific type of visualization or modelling application. A more thorough discussion of this library is given in Chapter 3.

The primary goal of scientific visualization is to provide systems suitable for use in research and development. A problem which users do encounter is to decide on the most appropriate visualization which will best highlight the desired properties of their data. In attempting to address this problem researchers have focused on systems, or approaches, that automate the visualization process, to some degree, based on the data

being visualized. Examples of this are the AutoVisual system of [Beshers 94] and the six approaches to automated visualization discussed by [Robertson 94]

Partially automating the process entails dividing decision making between the user and the system. The user has the power to disregard decisions made by the system and replace them with others deemed more effective. In fully automated systems the onus is entirely on the system to derive the most effective visualization for a given data set. Methods for this type of visualization creation are based on mathematical formalisms and concepts rooted in artificial intelligence. The value of the visualization created must then be determined by the system based on many criteria obtained from specification of goal, task of the visualization (exploration or presentation), user profiles and type of hardware on which the system is deployed.

It should be noted that the functionality of the systems discussed in this section and the goals they attempt to satisfy are not unique to any one particular system. The issues discussed, such as extensibility, ease of use and distributivity, are common to most systems, both Dataflow and non-Dataflow. The different systems attempt to address these issues in different ways and with varying level of success

2.4 Discussion

Throughout this chapter various modelling and visualization systems and paradigms were reviewed separately because of their required and distinctive functions within a modelling system.

To better understand numerical modelling **Maxwell™** and **Maple™** were examined as two representative systems of current state of the art.

The **Maxwell™** system represents a modelling system for users who do not wish to be involved with the internals of the system. These users require adequate functionality which is easy to find and use.

Maple™ represents a modelling system for use, mainly, by programmers as it requires knowledge of programming techniques and data structures. It is a valuable system because, as a symbolic and algebraic manipulation system, it supports the creation and prototyping of programs such as numerical models.

To better understand the field of scientific visualization two paradigms, Dataflow and non-Dataflow were discussed. Within the non-Dataflow category several existing systems were highlighted to illustrate their design.

Although these systems represent only a small segment of those which have been created, studying them illuminated different architectures and system designs which attempt to address a number of scientific visualization issues. Several of these issues are system and code extensibility, distributivity, generality (to other application domains) usability and code/module reuse. Equally as important was that this investigation revealed problems which must be addressed for future visualization systems as well as the incorporation of visualization functionality in modelling systems.

It is clear that no specific approach has lead to the "perfect" visualization. This is reflected by the lack of a visualization reference model and data model. This lack of a

reference model is an indication that more work is needed to identify and specify the role and function of visualization within a modelling system.

Together, the case studies discussed in this chapter illustrate desirable components of a computer modelling system. They are presented within the context of the **VPMS** system and include:

- The distribution of modelling system functionality.
- A structured interface which provides easy high-level access to functionality.
- Appropriate software abstractions which facilitate system openness, extensibility and provide a lower level of access to visualization and modelling functionality. An example is the **Maple™** library functions or the **Visualization Toolkit**.
- The appropriate abstractions for the incorporation of components, which have been created independently, into the system. An example is a numerical model prototyped using **Maple™**.
- A modular approach to system construction.
- The isolation of layers of specific functionality such as the visualization functionality of the **Visualization Toolkit**

Limitations of a system which should be avoided have also been highlighted: these include:

- Limitations or restrictions with respect to available functionality.
- Restrictions with respect to methods or processes of accessing functionality.
- Generality; a system should not be too general. It should provide application specific functionality (or context).
- A dependency of system modules or layers upon others. The term layer in this context does not refer to a software layer but a system layer which encapsulates a portion of the system functionality.

Based on the above identifications the design of a portion of a modelling system which is partially realized through the definition of software layers (libraries) is presented in the next chapter.

Since the introduction of scientific visualization, the field has been steadily growing as a scientific discipline. Although it is still relatively young, a growing body of research has evolved which contains the contributions of many scientists with respect to both theory and implementations (in the way of actual functional systems). Relevant publications for reference are the conference proceedings, *IEEE Proceedings of Visualization*, the *IEEE Transactions on Visualization and Computer Graphics*, and the journal *IEEE Computer Graphics and Applications*.

There is also an increasing treatment of this topic in textual format. Texts such as *3D Computer Graphics*, [Watt 93], *Scientific Visualization: Advances and Challenges*, [Rosenblum 94] and *Visualization in Scientific Computing*, [Haber 90], dedicate one or more chapters to the subject.

Chapter 3 SYSTEM DEVELOPMENT

3.1 Introduction

The focus of this chapter is the description of the design of a modelling system which encompasses modelling functionality and is applicable across multiple application domains. In this design, modelling and visualization components are segregated into categories and the necessary software layers which provide appropriate application specific and application neutral functionality are constructed within each category. In the current illustrative case, the functionality defines a portion of an electrochemical modelling system.

Through an examination of user requirements and a study of existing modelling and visualization systems and paradigms, a set of functional and non-functional requirements and system goals is identified. These are discussed below.

The functional requirements refer to functionality the system should possess which directly relates to the user. The non-functional requirements refer to elements of the system which do not directly concern the user; elements such as extensibility, portability, distributivity and code/system module reuse. This discussion is followed by a description of the system design.

The discussion of the system design is divided into two sections. The first section, the general design, provides an overview of the system including an explanation of system components, models and visualizations, and their integration within the entire system.

The second part of the discussion describes the use of three class libraries which enable the general design to be realized; these are: the Visualization Class Library, the Model Class Library and the Yoke Class Library. These libraries support the creation and extension of models and visualizations as well as provide for their integration within the modelling system.

3.2 System Requirements and Design Goals

The ultimate goal of **VPMS** is to provide a powerful and easy to use system which encapsulates modelling system functionality across a breadth of domains. To satisfy this goal, a set of functional and non-functional requirements is identified and described. The functional requirements are given in the context of the electrochemical application.

3.2.1 Functional Requirements and Goals

The functional requirements, which describe what the system is supposed to do, form a high-level core functionality of the system. The requirements below do not form a complete set; as the system evolves, additional requirements will be specified and incorporated into the design. Functional requirements are provided by the software layers and components within the system design and include:

- The ability to activate an environment (a window) which provides an interface to the modelling portion of the system. This environment should allow the selection of a numerical model and associated user interface allowing input to the model.
- The interface is to allow specification of a plate geometry via graphical, textual and mouse driven input metaphors. Sizes, positions, properties, orientations, materials, boundary conditions and any other necessary parameters must be graphically and textually editable.

- The system must allow the user to select from several different mesh types, i.e. rectangular and curvilinear, which can be superimposed on top of the plate geometry. The mesh specifies points on the geometry at which values are calculated.
- Once the parameters have been entered the model should proceed to generate data. The model should produce a sequence of intermediate data sets which can be visualized. This supports interactive steering to determine if computation is proceeding correctly. It should be possible to make adjustments to data to modify the computation.
- When a numerical model has completed its simulation, the user should be able to choose from a number of different visualization techniques, such as arrow plot or color plot, to visualize the data. It should be possible to provide multiple visualizations at once, both identical and different, for comparison purposes as well as for determining the visualization which is best suited for interpretation of data.
- The environment should provide support for the exploration of the data. For instance, functionality should include:
  - ➢ different color mapping schemes.
  - ➢ the use of different glyphs (graphics such as arrows).
  - ➢ the ability to bind different data values to different glyphs or elements thereof (i.e. magnitude is bound to the length of an arrow and electrical charge to its color).
- A user should be able to acquire or programmaticly create new visualizations (and models) and easily integrate them into the system for execution.
- The system should support distributed operation allowing distant users to have access to modelling system functionality regardless of whether or not they have this or any other modelling system.
- The user should have access to functional tools which allow analyses or secondary computations on data and have the results visualized. An example is the calculation of the electric field about the plate geometry from the computed electric potential values.

The above requirements represent high-level functionalities of the system. Each of these can be expanded to reveal a more detailed and specific set of requirements to be addressed; for example, the specific function to invoke when the file menu is activated. Detailed descriptions of requirements and functionality of all of the above requirements are not discussed due to the continual evolution of this research. The highest level

specification, however, is expected to remain the same, subject to the addition of other functionalities; in other words, extensibility of the system.

3.2.2 Non-Functional Requirements and Goals

The following subsections describe non-functional requirements and goals of the system. These goals represent functionality of the system which are of concern to the developer of the system regarding its structure and construction. These are core requirements which the system design addresses and satisfies. The design (structure) which results from the non-functional requirements, encapsulates the functionality identified in the functional requirements. The non-functional requirements include:

- Modularity and Extensibility, Portability and Reusability.
- A complete modelling system.
- Distributivity.

3.2.2.1 Modularity and Extensibility, Portability and Reusability

In order that modelling systems continue to provide assistance to users while application domains as well as the field of computer modelling evolve, it is necessary for them to be modular and extensible. Modularity and extensibility aid in ensuring that a minimum of effort is needed to update the system and that any changes be as localized as possible The object oriented paradigm use of abstractions, the modularization of components (environments) and the isolation of functionality are used to achieve a modular and extensible system.

The portability of any system can be enhanced through the use of standardized languages such as ANSI C or C++ as well as appropriate compilers which produce code

for different hardware platforms and their native operating systems. Additional portability issues concern native windowing systems. A first step in supporting this type of portability in this prototype, is the isolation of core system functionality from a particular windowing system.

In addition to the reuse of object oriented classes and abstractions it is also possible to reuse entire modules as illustrated by the Dataflow paradigm, the **FAST** and tool-based systems. An extension of this type of module reuse implemented in this design is the reuse of entire visualization and model environments (to be explained in Section 3.3).

The environments embody visualization or modelling applications such as an arrow plot or a numerical electrostatic model. Once created they can be reused in any version of the **VPMS** system, irrespective of the application domain. For example, an arrow plot visualization can be used in either an electrostatic or magnetostatic application provided the input data is consistent.

3.2.2.2 Complete Modelling System

It is a design goal of the **VPMS** system to provide a set of modelling system functionalities. More importantly, as stated in the introduction, the goal is to provide a system design supporting the addition of appropriate functionality required of a complete modelling system within the context of the specification of the electrochemical application as well as in different scientific domains with specifications similar in scope (i.e. magnetostatic).

The appropriate functionality refers to the availability of adequate numerical models for simulation as well as appropriate visualizations which aid in the interpretation of data. An example is the use of a color plot to examine computed electrostatic potential values. Additional functionality is represented by tools which perform secondary computation including the calculation of data such as capacitance or divergence of an electric field.

3.2.2.3 Distributivity

Through the use of networking technology, research and development efforts are increasingly freed from spatial and temporal restrictions. Computer networks such as LANs, intranets or the Internet allow researchers to share data, information, ideas, and discoveries.

Some visualization and modelling systems provide support for distributed visualization, as evidenced by the **FAST** and tool-based systems and the Dataflow paradigm. This support comes in the form of distributing data or computational modules on remote machine. For example, some systems distribute compute-intensive modules to remote machines which provide better computing performance. Data is then accessed wholly or partially.

Distributed visualization and modelling capabilities are useful in supporting collaborative research and development. The distributivity of this design is an extension of the type provided by the above systems. Instead of distributing small modules (which are a part of a visualization or model) **VPMS** distributes entire visualization or model

applications. An example is a color plot visualization environment which provides all color plot functionality (interaction and computation) as a distributed component.

3.3 System Design

The remainder of this chapter presents a design for a computer modelling system satisfying the functional and non-functional requirements and goals.

3.3.1 Data

To support modelling system functionality, the data structures, or objects, within a modelling system must encapsulate large quantities of data. They must also support an effective method of access to and manipulation of this data. An example of the requirements of these structures, or objects, can be illustrated by their implementation on different platforms such as a single large memory machine versus a cluster of smaller memory machines supporting a form of parallel execution. Different segments of data must be managed on each of the smaller machines, possibly with redundancies between them. Data on the large memory machine need only be managed as a single data set.

The issue of data representation is not addressed in this thesis because of the large scope of the field of data base research. For the purpose of the design and description of the system it is assumed that data is stored in the appropriate structures with adequate methods for accessing it, structures such as those in [Schroeder 92] and [Schroeder 96].

The implementation, since it is a prototype, uses simple data structures which facilitate the exploration of the system design. For example, a simple function is used to

simulate output from a numerical model. The output is represented as a one dimensional array.

Important future research, which is continuing in the field of modelling and visualization, is the development of appropriate abstractions for data.

3.3.2 General Design

The overall system design is similar to the tool-based paradigm of visualization. In that system, the visualization "tools" embody the individual visualization techniques such as arrow plot, cutting plane, contour plot and so on. Similarly the model "tools" are responsible for different numerical simulation of phenomena. The **VPMS** system differs from a tool-based system in the design, responsibilities and position of each of the tools.

In this design, the term "tool" is not entirely appropriate for the visualization and modelling components because they are not subordinate to other system control modules or layers. Their functionality and responsibility exceeds that of a simple tool. In a traditional tool base system the responsibility of a tool is restricted because it must operate within the context of other system components as seen in Figure 3.1.

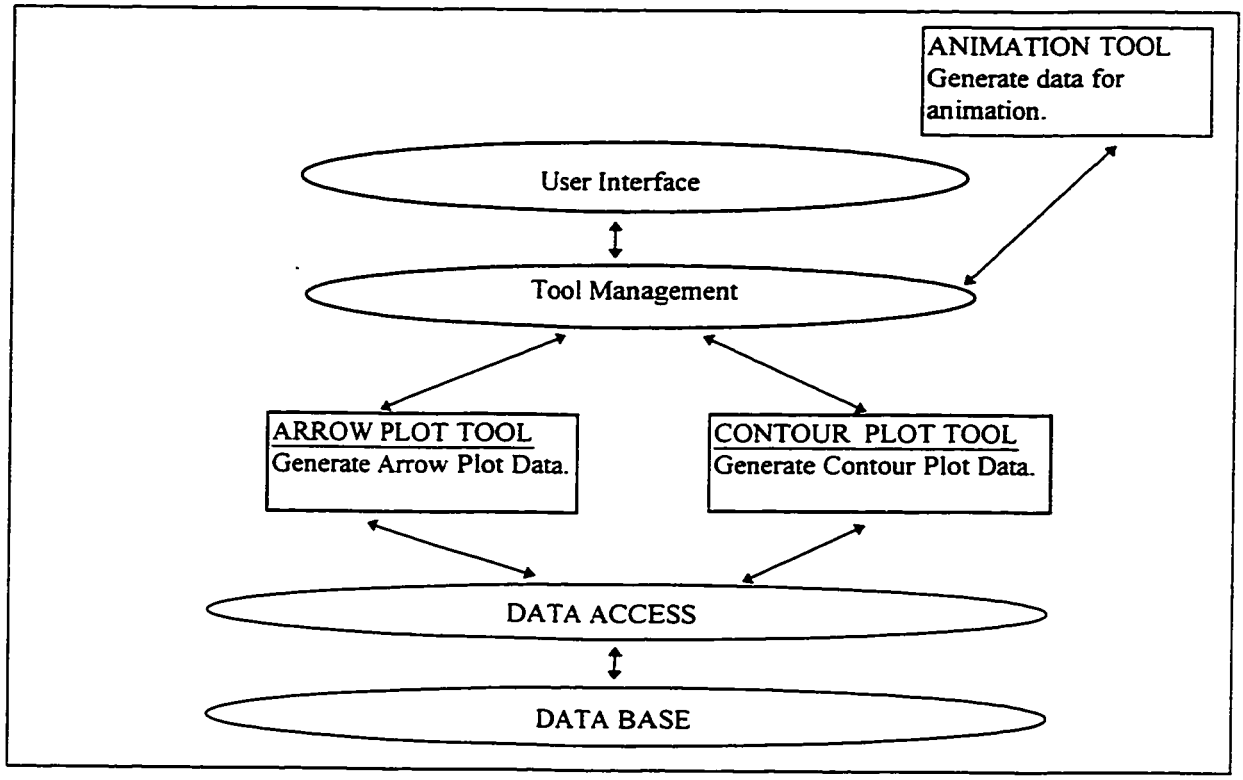

Figure 3.1: This figure represents the tool-based approach of [Brittain 90]. Tools depicted in the diagram (ARROW PLOT and CONTOUR PLOT) are tied to system components, namely the Tool Manager and the DATA ACCESS tool/layer. These layers provide common functionality for each similar tool.

The difference between the **VPMS** system and those illustrated by Figures 3.1 and 2.6 is that visualizations and models are designed as individual autonomous environments.

Individual environments are freed from restrictions imposed by other system components or layers. Visualization and model environments provide their own user interface so that a separate layer is not needed to intercept input which must then be sent to the appropriate environment. Environments also govern themselves and do not need to operate through other system layers to perform functions such as rendering or requests for memory and its management within the environment. This is possible because the mechanisms which provide this functionality are contained within each environment (via

the class libraries discussed in Section 3.3.4). Since much of the code in similar environments (e.g. visualization) is the same, redundancy exists resulting in increased space overhead. For example, two or more executing visualization environments will have many objects (object oriented objects) between them which are identical. This overhead is necessary to achieve the autonomy of environments so as to support the goals of the **VPMS** system. Figure 3.1 can be modified to represent our design as in Figure 3.2.

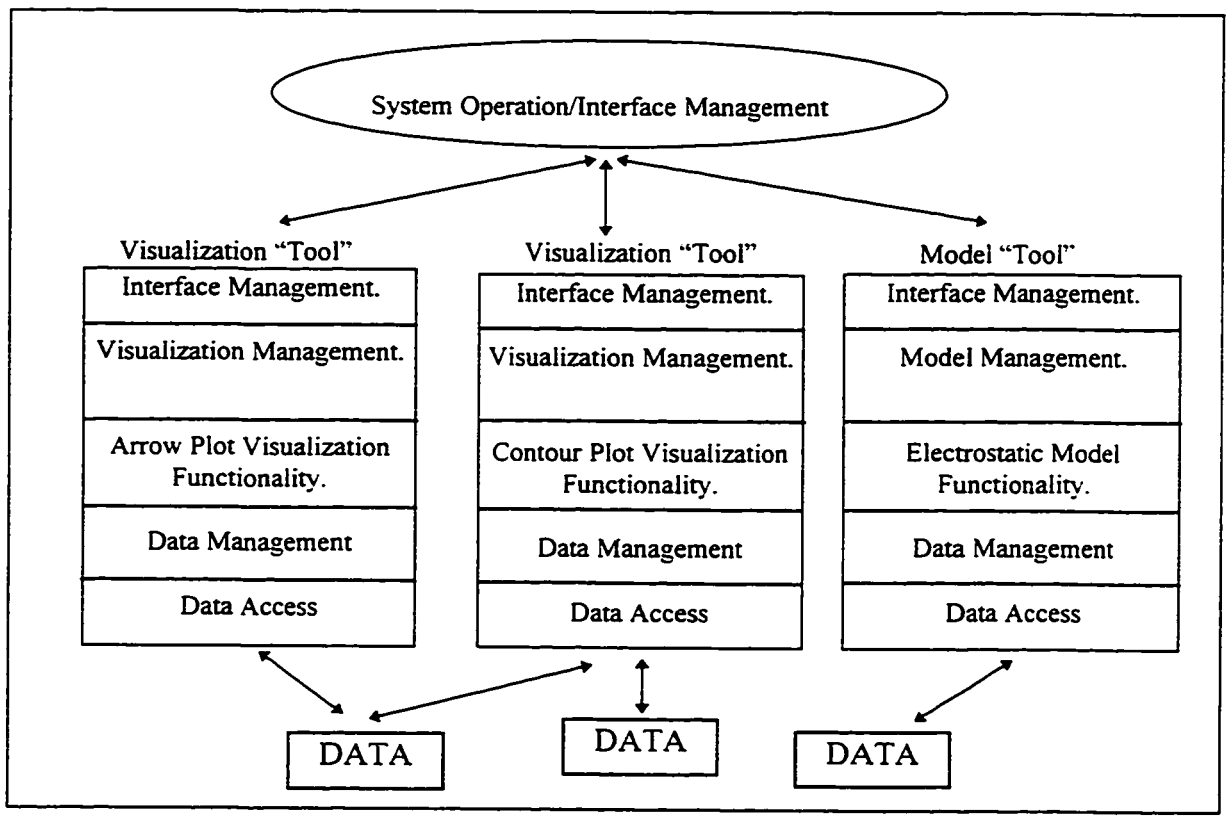

Figure 3.2: In this design the layers of the system in Figure 3.1 have been absorbed by the visualization and model environments. The abstraction for the functionality of each environment is provided through the object oriented paradigm. This is a functional, as opposed to system design, abstraction and it is encapsulated within each environment.

Figures 3.2 illustrates the encapsulation of the necessary control functionality which provides autonomy for environments. It also demonstrates that they are not totally

independent because there is communication with other parts of the system (this limitation on the term independent is maintained throughout this paper). Layers of software abstraction (provided by the libraries discussed in Section 3.3.4) are contained within each environment.

Since environments contain the functionality which controls their operation, a change to any one of them does not effect the others. Additionally, their autonomy enables their incorporation into the system without adversely affecting pre-existing environments and requires no modifications to other portions of the system. For example, new or modified environments can be rendered along with others in a single window to create a composite image, or be rendered to an externally spawned window to provide two versions of the same visualization as in Figure 3.3.

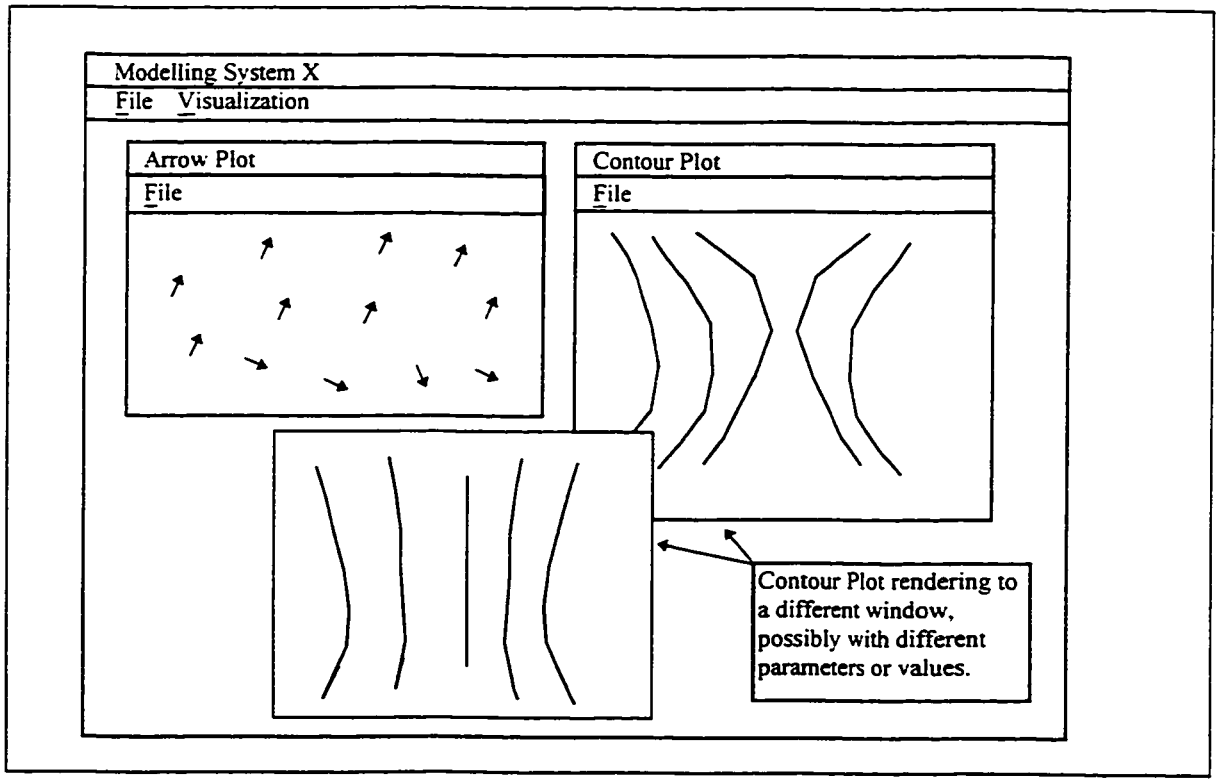

Figure 3.3: A Contour Plot visualization modified to draw to an external windows. This modification need not affect any other visualization, i.e. the arrow plot.

The individuality of the environments allows them to be configured such that each can be optimized according to its visualization type (arrow plot, contour plot, cutting plane). Definitions for unique interfaces are self-contained, thereby allowing the context for input to be tailored to specific visualizations or application domains which provides preferential contexts for input (via language , notation and other formalisms).

In addition to the benefits of modularity and extensibility offered by self-contained environments, there is an added benefit of distributivity. Many systems deal with distributivity of operation by providing remote data access or allowing component modules to execute remotely. Modules which execute remotely act as one part of the entire system. For instance, a particular module may run on a super computer, generate geometric data and return it for rendering on a simple PC, terminal or workstation.

A new approach to distributivity of modelling system functionality, as a result of this design architecture, is that as self-contained environments are able to operate independently, outside of, and disconnected from the system environment. As a result visualizations can be created and retrieved through a network and executed on a local machine without the benefit of the entire system implementation.

An implementation of this sort allows researchers to share data, visualizations and numerical models. Another ramification of this design is that new visualizations and models may be created by either the designers of the system or programming knowledgeable users. New models and visualizations serve to extend the system or make it more application specific. Assuming a specific protocol is followed they may be

plugged into the system and used. The protocol takes the form of a class library used in the creation of a visualization or model and is discussed further in Section 3.3.4.3.

A visualization or model that operates external to the system environment will not benefit from specific functionality which is defined within the system environment. For example, a visualization can not render to different destinations or operate in conjunction with other visualizations as this requires system environment specific functionality.

All of the functionality specifically implemented in a model or visualization environment is still accessible while they are operating outside of the system environment. Also, when operating outside of the system environment, the user interfaces do not change. For example, functionality which is implemented in a visualization includes the ability to change color mappings or substitute arrows for some other appropriate graphic, or glyph.

The design supports collaborative research because visualization environments and models can be distributed outside of the system to other researchers who can perform the same experiments using the same environments and view the same results. Additionally they can make subtle changes, in the modelling or visualization process, to test their own hypotheses. It is also possible to distribute environments to users who have a **VPMS** system. The environments may executed within the system of that particular user. The following figure illustrates a more detail design of the system.

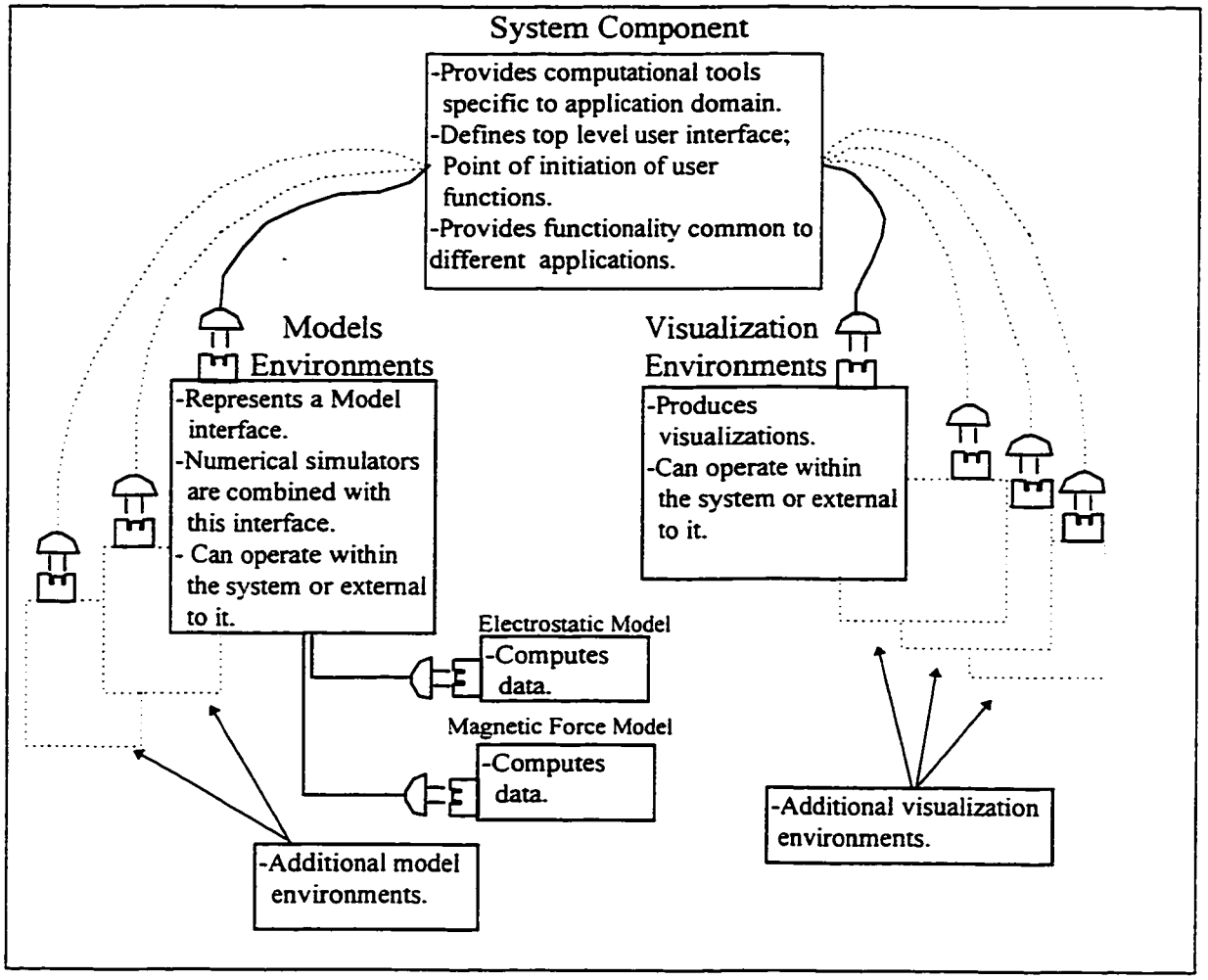

Figure 3.4: A high-level system diagram. It illustrates the System Component to which multiple models or visualizations can be attached. Each of the environments can operate as part of the system or independently, outside of the system environment. The connections to the System Component, as illustrated by the "plugs", is provided by the Yoke Class library. Note that different models can be combined with a single model interface or front end.

3.3.2.1 Network Distribution

[Wood 96, p. 82] describes several scenarios for accessing visualizations over the *World Wide Web*. One of them, identified by the section title "**Scenario 3 - The viewer creates the visualization**" places the responsibility of creating the visualization on the viewer, or the remote user, who wants to visualize some data. In this scenario the user

can retrieve the desired data from a remote server and use some current visualization package, running locally, to interactively explore it.

[Wood 96] notes some disadvantages to this type of scenario which are valid, but addressable. They are:

- The appropriate visualization software must be available to the user.
- The visualization software must be able to read the data in the format in which it was retrieved.
- The user must have the appropriate processing power available to use a visualization package.
- The user must possess the knowledge to use the visualization software.

The method of distribution of **VPMS** is able to address the first three of these problems, thereby making it a viable solution for distributed computer modelling.

To address the first problem it is assumed that the same people who are interested in specific data, numerical models and visualizations are those who are actively involved in research and development within the field to which the data and visualization or modelling capabilities pertain. For example, those who are interested in electrochemical research will clearly be the ones most interested and in need of the appropriate data, numerical models and visualizations concerning electrochemistry. Given this assumption it is reasonable to conclude that these researchers would want to have access to modelling systems specific to their area of research which can satisfy their needs. If these are readily available they will likely become tools of choice.

The system design encompasses the necessary tools for research and development. It allows researchers to retrieve models and visualization environments which can execute on a local machine, either inside or outside of the system environment,

and provide interactive data generation and visual analysis. Therefore, access to the data and appropriate tools for modelling and visualization are available via the **VPMS** system. The user does not have to purchase or already possess a correct modelling system because portions of an appropriate one can be retrieved.

With regards to the second of the issues, data incompatibility, if both a model and a visualization are designed specifically to operate within the **VPMS** system then the data format for model output and visualization input will be consistent. Therefore, the user can either access the data which a model, from the **VPMS** system, has produced, or acquire the actual model and have it execute locally and in tandem with a retrieved visualization from the **VPMS** system.

It is evident that the power, speed, available memory and graphics capabilities of a computer directly affects the execution of a modelling system. Computational power is an issue in the execution of large numbers of complex mathematical operations. The low end platform for implementation of **VPMS** is a Pentium™ class machine, although it is constructed for portability to more powerful **UNIX™** based platforms (**Sun™**, **SGI™** or **IBM™** RISC machines). Although the PC does not yet possess the processing power of a workstation, it still provides an adequate platform for modelling of the type described in this paper.

The fourth of the issues highlighted is specific to end-users (as opposed to programmers) of the system. It states that they must have knowledge of a particular visualization software system. Potential users who require visualization functionality will, eventually, have to learn to use some visualization system, whether it is the **VPMS**

system or another. Possibly, the only way to address this issues is to make the system as easy to use and learn as possible. The fact that ease of use and speed of learning are subjective makes this issue difficult to address. With respect to programmers, the development of increasingly higher level CASE tools can aid in the design, creation, assembly and eventual execution of modelling system functionality. A discussion of CASE tools for modelling is beyond the scope of this thesis.

3.3.2.2 Security

The type of distributed interaction discussed in this section does not support any kind of security. There is no way to ensure that a model or visualization environment that is retrieved over a network will not adversely affect the local system. This problem, however, can be partially addressed by only performing this type of interaction (collaboration) with a trusted software provider, colleague or research and development firm, to name a few.

In the **VPMS** system, security will remain an issue until visualization and modelling environments can be exchanged universally in a secure fashion. Implementing the system, or parts thereof, in a programming language which supports security, for example **Java**, will aid in achieving this goal. **Java** Is a platform independent, object oriented programming language which is used for the creation of *applets* for use within *World Wide Web* browsers as well as common computer application programs. The importance of an *applets* is that it executes on the host machine in a secure environment. The degree of security can, further, be relaxed by the user.

3.3.2.3 Use of Java

In the statement of future research [Wood 96] validates a system in which **Java** is used as a remote interface to a visualization system. This functionality is also a design goal for **VPMS**.

To provide secure remote access to visualizations and models, the framework of the **VPMS** system supports the distribution of interfaces to visualization environments and models as opposed to the distribution of the actual environments. In this mode of execution the visualization, or model, executes within the computer modelling system on a server. Remote users access the functionality of a visualization or model from a distant computer by retrieving an interface to them.

The interfaces execute as *applets* within a **Java** enabled web browser on a remote machine and are used to dynamically enter parameters and configure the visualization, or model, using graphical and textual input. This scenario is illustrated in Figure 3.5.

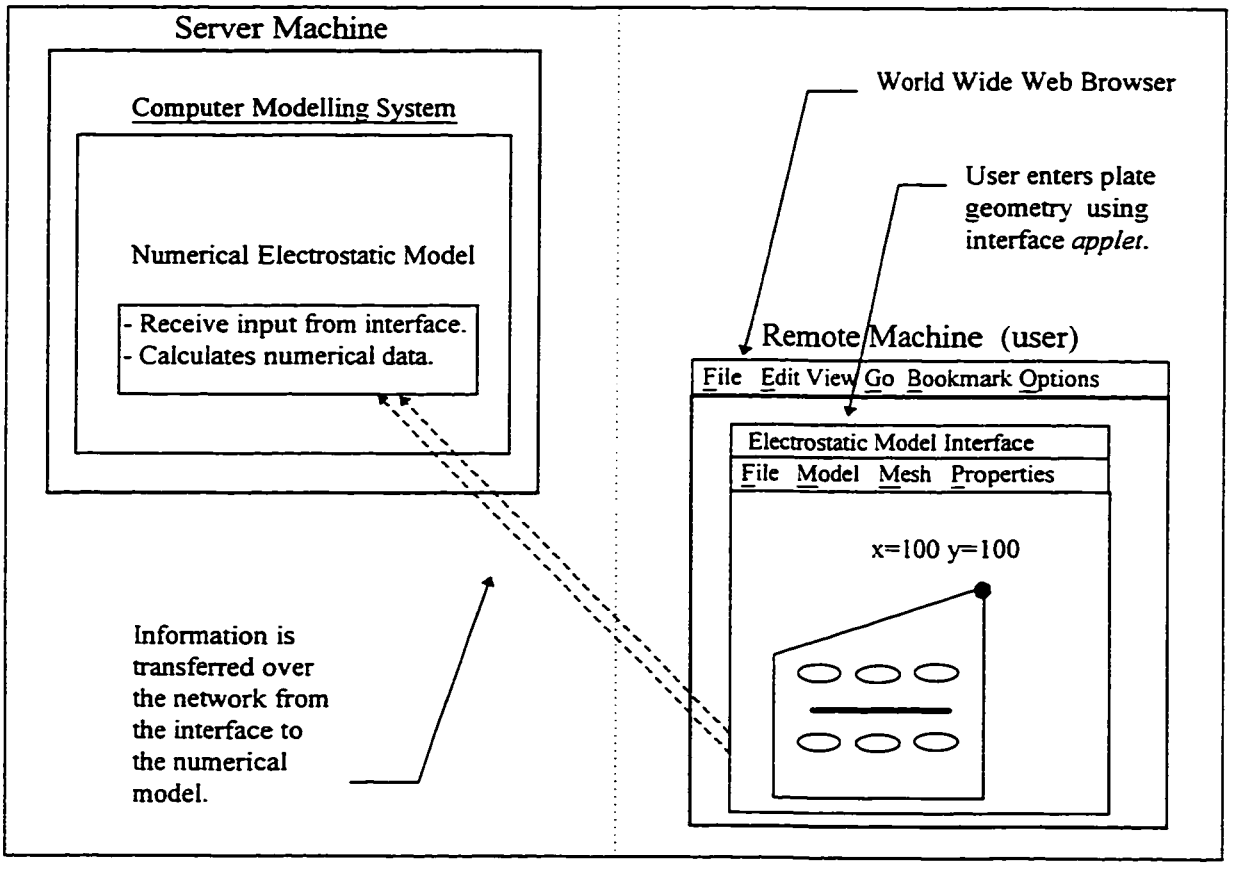

Figure 3.5: This figure illustrates the user interface to a model executing as an *applet* in a web browser on a remote machine. Once all parameters are entered information is sent to the numerical model of the system on the local machine. The use of a visualization is achieved in the same manner.

When a user has completed the input of data, the information is transferred back to the modelling system on the server system which then executes the visualization, or model, using the parameters supplied by the remote user. If a visualization has been produced a graphical image can be sent back to the user.

If a model is accessed then the user can retrieve the computed data or explore it remotely, using the method just outlined.

The present state of networking technology can introduce several drawbacks to this type of interaction. Possible results of the constant transmission of information

include corruption of data, interruption of transmission and common network delays, to name but a few.

3.3.3 System Component

The responsibility of the System Component is to provide a framework, or environment, in which visualization environments and models can function as well as cooperate with each other. The System Component can be seen in Figure 3.4

The System Component contains the functionality unique to a particular field of study. For example, the calculations of **E** and *mag*(**E**), are contained and execute within the System Component and serve to define it as a domain specific system. Control and access to these functions is provided by the System Component and its interface. Relevant information in the System Component, such as results of computation, is made available to the visualization or model environments by the communication established through the Yoke class library. Information may also flow into the System Component.

The System Component framework is also responsible for coordination of the multiple visualizations as well as models which may be operating within the system. Access to the functionality of each visualization is provided by the visualization environment itself while activities such as selecting a particular visualization, combining it with others, adding new ones to the system or removing them is handled through the System Component. The System Component framework communicates the desired action and any relevant information, via the Yoke library, to the appropriate visualization. This information may take the form of the proper window in which to render.

Responsibilities of this framework also include the distribution of visualization environments, models or **Java** *applets*, receiving information from an *applet* acting as a remote interface to a visualization, and passing the data onto the appropriate visualization. The same is true for model environments.

Keeping domain specific functionality within the System Component of the modelling system eases the burden of migrating the system to another application domain because many of the necessary changes are restricted to the System Component.

Items such as the functional calculation tools, for secondary computation, are defined to operate within the confines of the system enabling the visualization environments, and their creators, to concentrate on details pertaining to visualization and removing the burden of dealing with additional functionality which can change depending on the nature of the visualization system. This supports the portable, reusable and distributable design criteria.

3.3.4 Design Specifics

In this section we establish a design for three class libraries. These libraries and the definition of their functionality constitute the contribution of this thesis (with respect to design) to the field of computer modelling. They encompass the layers of software abstraction necessary to achieve the system design and its functionality within the confines of the electrochemical application. To facilitate construction of both functionality and abstraction, the object oriented paradigm is used. This paradigm also supports extensibility, modularity and code reuse of the system and its components.

The first library, The Visualization Class Library, contains the classes which facilitate the creation of independent visualizations. The second library, the Model Class Library, is used for the creation of models and encapsulates functionality for both interface creation as well as functional (numerical) components of the model itself. The third, or Yoke Class Library, facilitates the integration and disintegration of visualization environments and models from the system. These classes handle communication between models, visualization environments and the system.

3.3.4.1 Visualization Class Library

This portion of the thesis identifies layers of functionality and abstraction required to create visualization environments and isolates them within the Visualization Class Library.

The library must contain a layer of functionality which accepts numerical data and transforms it into a visual representation. This layer must perform tasks such as interpolation, integration, geometric computation, and mapping of data to graphical output primitives. This layer also provides access to underlying rendering capabilities.

The Visualization Class Library must support the handling of data which makes it available to appropriate functions (such as those mentioned in the previous paragraph). Data manipulation must also support functionality such as interactive steering or visualization of results of intermediate computation produced by numerical models. This facility allows segments of data identified through visualization to be isolated, modified and re-entered into the system for continued processing.

Additional layers of Visualization Class Library encapsulate functionality for direct communication with other environments (both visualization and model) through the availability of necessary data structures and method invocations. For example a visualization environment may exchange data directly with a model environment (e.g. to support interactive steering).

Recently [Schroeder 96] reported an object oriented toolkit for graphics and visualization known as the **Visualization Toolkit** (or **VTK**).

Although it is not used in the implementation, **VTK** represents the portion of the functionality of visualization environments concerned with the transformation of numerical data into graphical images and the rendering of the result. Like **Maple™**, **VTK** provides primitives (library functions) which are used to implement this functionality. Enabling **VTK**, or any other similar library, to work within **VPMS** requires the Visualization Class Library to provide the functionality which **VTK** (or similar libraries) do not, for example **VTK** does not provide functionality for data manipulation required by **VPMS** (e.g. to support interactive steering). The Visualization Class Library must also provide appropriate methods and abstractions which provide the linkage between itself and **VTK** .

The integration of an external library, such as **VTK,** into the Visualization Class Library requires that the Visualization Class Library possess abstractions which deal with input and output of data to and from the external library. For example, **VTK** requires data in a particular format, therefore, the Visualization Class Library must supply abstractions for the transformation of data from a native (**VPMS**) format into the format

required by **VTK**. With respect to output, **VTK** renders to internal abstractions of windows. The Visualization Class Library must provide abstractions which bridge the gap between the representation of rendering windows in the **VPMS** system and the representation in **VTK**.

Through the Visualization Class Library, which contains appropriate software layers and abstractions, the goal of enabling the creation and operation of a visualization environment as discussed in this chapter are satisfied. The isolation of visualization functionality aids in the abstraction and creation of visualization environments because other issues (modelling) need not be dealt with directly.

3.3.4.2 Model Class Library

This library provides the functionality for the creation of model environments. The environment includes numerical models and user interfaces for input.

The classes for the creation of user interfaces encapsulate methods needed for the creation, configuration and control of graphical objects and their attributes. For example, a typical electrochemical model may require a plate geometry about which it calculates potential values. The plate geometry consists of graphically represented physical elements which have various properties assigned to them, including material of construction, boundary conditions, position and voltages. Figure 3.6 represents a graphical interface containing plate geometry elements, to which a model is associated.

The library is responsible for the abstractions which can encapsulate numerical models so they may be incorporated into the modelling environment. This abstraction

provides a standard interface between the user interface and a numerical model in addition to an encapsulation of input data and the methods required to make the data available to a numerical model.

Additional classes include those which provide building blocks for creation of numerical models, classes such as those for the generation of meshes. These classes are typically computational in nature, reflecting the kinds of functionality found within the **Maple™** library, for instance.

This library isolates the modelling portion of a computer modelling system. This supports the separation of environments, focuses development specifically on modelling functionality and eliminates dependency of modelling functionality on other system components or layers.

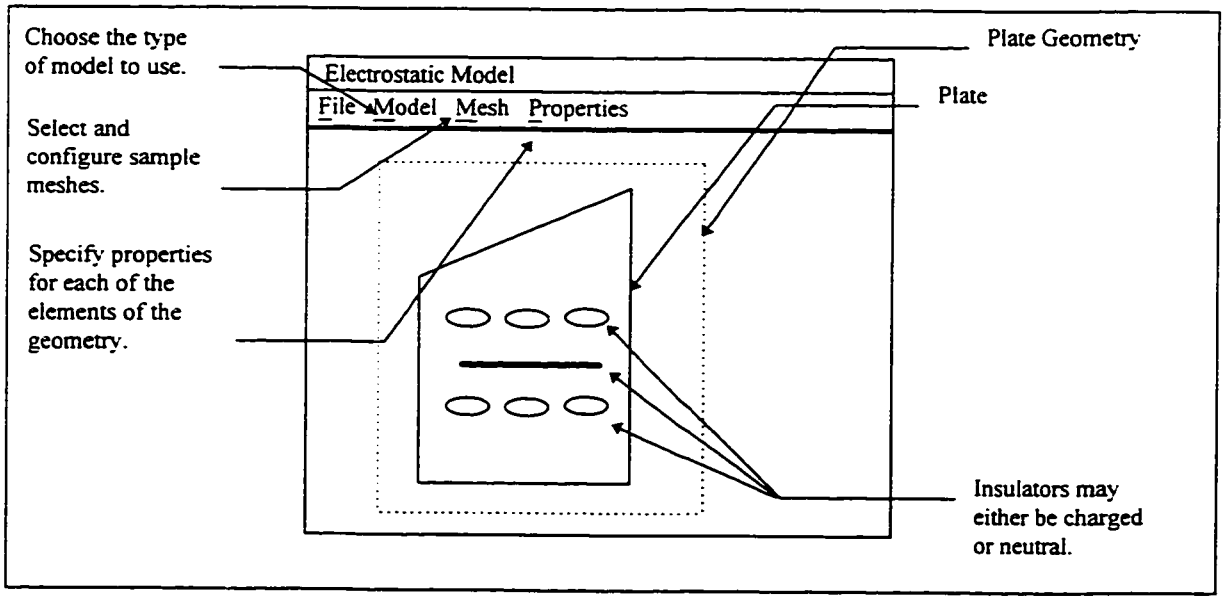

Figure 3.6: This diagram represents a graphical input module which is used to specify parameters and a plate geometry for input into a numerical model. A specific model can be selected which will use the parameters to calculate data such as potential values, in the case of the electrostatic model.

3.3.4.3 Yoke Class Library

Visualizations and models which operate as individual environments, as described in Section 3.3.2, require functionality for communication with the System Component. This portion of the thesis identifies this functionality and encapsulates it in the Yoke Class Library.

Visualization, or model environments inherit functionality (in an object oriented sense) from the hierarchy of classes encapsulated in the Yoke Class Library. The functionality in the library facilitates the communication between an environment and the System Component allowing environments to connect to the modelling system. In other words, the library provides a "software handshake" between environments and the System Component. For example, when an environment is invoked (from the System Component), the System Component must send a message to the appropriate environment informing it to begin execution. Additionally, any information which must be passed to an environment from the System Component, such as information required to complete the connection between the two, is passed via the Yoke Class Library. For example, information an environment needs to begin execution, such as window handles or object references.

Pragmatically, this is accomplished through the definition of functions defined in the Yoke Class Library which the System Component calls. Conversely, functions within the System Component are made available to environments through encapsulation by the Yoke Class Library.

Functionality which provides environments with references to other environments, so that communications may be established between them, is also provided by the Yoke Class Library. For example, a visualization environment can query (through methods inherited from the Yoke Class) the System Component for a reference to the currently executing model environment. This is required so that the graphical input to the model, i.e. a plate geometry, may be rendered along with the visualization output. Typically, this graphic would be reproduced with the visualization of the data in order to provide a context for interpretation as illustrated in Figure 3.7. Note that the functionality for retrieving the data from the model environment is defined in the Visualization Class Library.

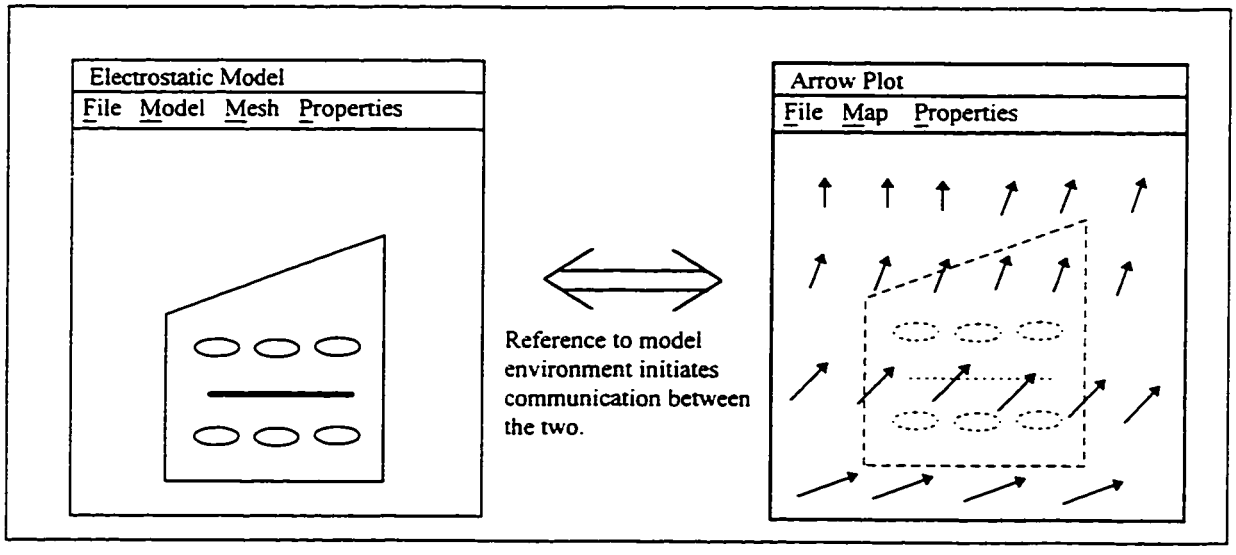

Figure 3.7 This diagram illustrates the required communication between a model and a visualization. This communication is established through Yoke Class functionality but the actual transaction between the environments is provided by functionality defined in either of them.

This is by design so as to isolate visualization and model functionality within their own environments. Therefore their creators do not need to deal with these issues because they are implemented in the Yoke Class Library.

The responsibilities of the Yoke class includes recognizing the mode of operation of a visualization, whether it is operating within the system environment or external to it. If operating independently then the Yoke framework is responsible for initialization and initiation of a visualization environment.

3.4 Discussion

The discussion of the system design presented in this chapter began with a specification of the system requirements. A general design which supports the goals of interactive scientific analysis, distributivity, modularity, extensibility, code and module reuse was given next. The notion of individual self-contained environments was described as one method for facilitating these goals. The environments encapsulate functional abstraction that is implemented as system layers of a tool-based system (see figures 3.1 and 3.2).

The general design was followed by a discussion of the detail design of the system. The detail design described the use of three class libraries. These three libraries form the basis for the functional abstractions of each of the environments (they are contained within the environments). They define both the abstract modelling system functionality in addition to the application specific functionality for an application domain

The reasons for the system organization as it has been presented throughout this chapter include:

- The desire to reorganize the functionality of a computer modelling system through the creation of environments for modelling and visualization.
- The desire to provide a distributed modelling system which can be accessed by users who require all or part of the functionality offered by the system.
- The desire to create reusable elements such as visualizations and models. .These elements can be reused in other versions of **VPMS** which lack one or more of them.
- The desire to create a system which can be extended by its creators or its users. This provides a method which allows the system to span other domains or to increase the usefulness in a particular application domain. This is achieve through the use of the three class libraries.
- The desire to create a complete modelling system by providing an environment for numerical modelling and an environment for visualization of results in addition to functionality for secondary computation of data based on output from models.

Toolkits such as **VTK** permit groups of users to create single purpose visualization applications which serve their immediate purpose. The result of this is uncoordinated and random visualization and model system development. On the other hand, a system, such as the one described in this chapter, make it is possible to reduce the need for creators of visualizations to think and design on a system development level and, thus, allow them to concentrate on a visualization or model development level; that is a higher level of abstraction. As a result, new visualizations and models are created which work in a standard modelling system as opposed to the reinvention of new computer modelling systems which have an intersection of functionality. In this respect, the **VPMS** system may also serve as a standard system, as discussed in [Butler 93], which can be modified and extended through the addition and deletion of layers of modelling system abstraction and of specialized components such as functional calculation tools, model and visualization environments.

In the next chapter a description of a specific implementation of the system design and initial layers of modelling system and application specific functionality is given.

Chapter 4 IMPLEMENTATION

4.1 Introduction

In this chapter we describe the current implementation and functionality of the **VPMS** system within the context of the revised design presented in Chapter 3. Suggestions for additions and modifications to the current implementation are discussed in Chapter 5 as part of future work and in Appendix B. Appendix A describes class structures of the current implementation.

As previously stated the current system implementation is intended as a prototype and representative of the initial design. Much of the functionality of the revised design exists in the current (initial) implementation, however additional functionality and restructuring is required.

Section 4.2 deals with the choice of **C++** as implementation programming language. Included is a discussion on the debate as to the level of object orientedness of **C++** and issues relating to portability of the system. Following this is a description of the implementation of the visualization environments and models as Dynamic(ly) Link(ed) Libraries. The class libraries which support the creation of models and visualizations are then introduced and described followed by a discussion of system efficiency.

4.2 Implementation Language

The programming paradigm for the implementation of **VPMS** is the object oriented paradigm. The reasons for this choice are well documented in literature. It is

sufficient to state that the object oriented paradigm supports the design goals of this system as well as methods for abstracting, encapsulating and objectifying modelling system layers and functionality.

The programming language chosen for the implementation is **C++**. The popularity of the **C++** programming language has resulted in the availability of **C++** compilers for all major computing platforms. The proliferation of this language ensures the portability of **ANSI C++** compliant programs.

C++ shares a tight binding with the C programming language and this relationship provides access to operating system functionality such as system commands, direct and dynamic memory access and allocation and access to network connections. The **C/C++** binding also provides access to many external libraries used for application development (increasing the support for and portability of **VPMS**) These libraries include graphics and rendering libraries such as the **Visualization Toolkit**, **PEX/PHIGS**, **OpenGL™**, **OpenInventor™**, etc., windowing libraries including **Motif™**, **OpenLook™**, **MFC™** and the **Win32™** API. Using functionality defined in **C** violates object oriented programming tenets unless this functionality is encapsulated within objects.

Additionally, it is also possible for other languages to access functions defined and written in **C/C++**. For example, **Java™**, through the use of native methods, can access compiled **C/C++** code. This final point has relevance to future work using **Java** discussed in the next chapter.

Another reason for the use of C++ is its runtime efficiency. The compute intensive nature of computer modelling makes execution speed and efficiency essential. The compiled nature of C++, and the fact that it retains the runtime efficiency of C, provides quicker execution than many other languages, particularly those that are interpreted.

The object oriented nature of C++ does, however, impose performance penalties such as the copying of data between objects, the use of methods to access data and the construction and destruction of objects. It is possible, however, to improve efficiency of execution of the system, as discussed in Section 4.6.

A final advantage for the use of C++ is the availability of a large number of class libraries for use within an application including libraries for data structures, mathematics and rendering.

4.2.1 Object Oriented Paradigm

Some computer scientists and programmers who use languages such as **Smalltalk™** and **Eiffel™** do not regard C++ as an object oriented language, and definitely not a pure object oriented language. According to [Booch 94], however, C++ is an object oriented language because it satisfies the following criteria:

- C++ supports objects that are data abstractions with operations for manipulating the local state.
- C++ objects are instances of a class.
- A class inherits attributes from its superclass.

C++ is not considered to be pure because it is a composite of an object oriented and an imperative language. As a result of this composition C++ allows for entities, for

example functions and variables, which exist outside of objects. This violates the object oriented principle as elements are present which are not objects.

From the point of view of an object oriented purist this is undesirable because the use of C++ can lead to errors and side effects and make it difficult to read, extend and reuse code. An alternative view is that C++ offers creators of software systems the benefits and freedom of using both an object oriented and imperative language such as speed and efficiency. This implementation uses C to perform low level memory allocation (malloc or calloc). These allocations are performed within objects and eliminate the need for the creation and management of additional data objects which consume system resources such as memory and CPU cycles.

4.2.2 Portability

As mentioned above, adherence to ANSI C++ improves the portability of the system due to the availability of C++ compilers for numerous platforms. Other important implementation aspects of the system which affect portability are windowing systems and rendering libraries. As a prototype, portability within this implementation begins with the isolation of kernel (or core) functionality.

4.2.2.1 Windowing Systems

To facilitate the use of different windowing systems, for example, Motif™ on UNIX™ or Win32™ for Microsoft™ operating systems, the core functionality of the system is isolated as much as possible.

Modules in the system that interact with a user via a graphical user interface rely on one or more functions to retrieve data from, or respond to the action of an interface element (e.g. text being entered into a text box). Essentially, these functions are window library dependent and are connected, or hooked, to the user interface components, such as buttons and text boxes.

These functions may be implemented within an object to operate together or implemented independently. Regardless of their specific implementation, they represent intermediaries which facilitate communication between window interface elements and system functionality. They receive data in the format which is defined by a user interface, convert it to an appropriate format, if necessary, and give it to core functions or objects which carry out the action requested by the user, such as the retrieval of a file. When a change of windowing environment is required, only the intermediary functions (or object which contains these functions) need be changed.

For example, the system requires input as a string object. A text box element in which the user inputs the text makes it available as a simple text string. This results in a type mismatch. Converting from string to string object in the core function binds this function to a particular windowing interface element and it must, therefore, be modified every time a new windowing system is used and in every part of the program which references the text box. This is shown in Figure 4.1.

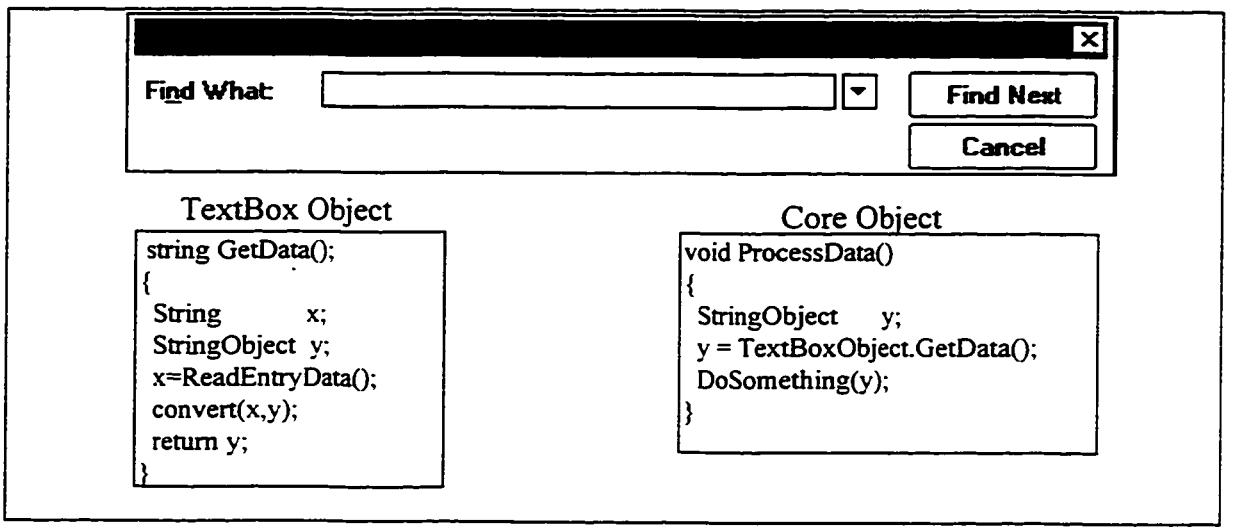

Figure 4.1: The TextBox Object is a class object which encapsulates the text entry dialog. The TextBox Object retrieves the text from the text box dialog, via the "ReadEntryData()" function, converts it into the proper format and returns it to the core object which requested the data via the "GetData()" function. Only the TextBox Object need be changed for a change in interface.

By abstracting the specifics of a windowing system element into an intermediary object or function, it is a simple matter to alter or replace this object or function quickly and without the possibility of introducing errors into the core function. This also speeds up the porting process as only one object, or function, need be changed.

4.2.2.2 Rendering Libraries

An important aspect of a modelling system is its ability to use different libraries. Because different computer platforms make use of different rendering libraries the portability of a system is directly affected by its ability to use these libraries. Within **VPMS** the ramifications of this are two-fold.

As a system strictly for end-users, it may be staticly compiled on the user platform using a particular rendering library such as **OpenGL™**. Because it is staticly compiled

the required rendering functionality becomes a part of the system executable and therefore the user of this system does not need to possess any particular rendering library.

If users intend to modify and extend the system through the use of the Visualization Class Library, the issue of rendering library support becomes more critical. The modelling system should support the particular library which is available to the user.

As a prototype, the **VPMS** system addresses the issue of portability of rendering libraries by using **OpenGL™**. **OpenGL™** is a library which is available for many different platforms and takes advantage of available hardware configurations for optimized performance. It is available for major computing platforms including **Windows95™**, **WindowsNT™**, **IBM RISC 6000™**, **SGI™** and **SUN™** platforms.

Additional portability can be achieved in the future by creating a layer of abstraction which encapsulates the operation of specific rendering libraries.

4.3 Visualization and Model Environments

Both visualization and model environments are designed to work within the system environment as well as external to it. To facilitate the operation of an environment outside of the sphere of the system, the visualization or model itself must be created, and compiled, as an executable application which can be invoked by the user. This supports the distributed use of the system as well its extensibility and component reuse.

It is also required that these environments be loaded or attached to the system, through user commands, while the system is running. To accomplish this both

visualization and modelling environments must be compiled as dynamic(ly) link(ed) libraries (DLLs). It should be noted that environments have not yet been compiled as DLLs in the implementation. However, experiments regarding the compilation of environments as DLLs have been conducted and research into the operation of DLLs performed. The use of DLLs is, therefore, deemed viable based on the results of this research which is summarized below:

- The DLL code is mapped into the address space of the running process which invoked it. This makes the executable code of the DLL available to the calling process.
- Any handles opened by the DLL are available to the threads of the process which called the DLL and vice versa. (**MFC™/Win32™**)
- Memory allocated by the DLL is in the address space of the process which called the DLL.
- Globally declared variables in the DLL are available for read and write operations to the threads of the process which called the DLL.
- Many processes can use a single copy of a DLL simultaneously allowing multiple instances of the same visualization or model. This reduces the amount of swapping required and saves memory.
- New environments can be created and added to the system without having to recompile the system. Only the visualizations or models themselves need be compiled.
- Shared memory is available for use with DLLs.
- With DLLs the only runtime overhead is when the DLL is being loaded and during its initialization.
- When the visualization or model is exited it can be removed from memory, freeing up the space it was occupying.

Dynamic(ly) link(ed) libraries may either be linked into the system at load time or runtime. In this system, runtime dynamic linking is used and it refers to the process of retrieving the appropriate DLL and mapping the code into the address space of the calling

process. It is the responsibility of the Yoke Class Library to handle the details of the linking with the System Component (refer to Figure 3.4).

The list of properties of DLLs given above hold for **UNIX™** systems as well as **Windows™** based systems, therefore the issues regarding the use of DLLs on different platforms is of a syntactic nature. For example, in a **Windows™** environment the "LoadLibrary()" function is used to load a DLL into the address space of the running process, while in a **UNIX™** environment the analogous function is "dlopen()". Under **Windows™** the "GetProcAddress()" function is used to get the starting address of a particular function in the DLL which is required to execute. To perform this under **UNIX™** the "dlsym()" function is used. The differences in syntax may be resolved by abstracting the functionality into software layers which provide a standard syntax.

4.3.1 DLL versus IPC

Environments can be implemented using Interprocess Communication (IPC) as opposed to DLLs. Several forms of IPC exist, including named and unnamed pipes, message queues, shared memory and sockets. Not all forms of IPC are available, nor are they implemented identically, on all systems. The most common types of IPC are pipes, shared memory and sockets. Essentially, IPC is used to overcome the fact that separate processes which communicate are in separate code spaces.

DLLs provide a better solution than IPC because environments are run in the same code space as the entire modelling system. Therefore they have access, not only to data, but to objects (references) and object methods allowing functionality to be executed

directly through object method invocation. Implement this type of functionality using IPC would require additional layers of abstraction to interpret the data which is transferred through a particular form of IPC. For example, data received from a process must be interpreted as an object method invocation.

4.4 System Component

The System Component (Figure 3.4) is a part of the modelling system which contains both application specific and application neutral functionality (discussed in Chapter 3). The System Component represents the toplevel environment with which the user interacts. It generates the main window and menu of options from which the user begins all modelling and visualization sessions.

From the main window the user activates a modelling or visualization environment. These actions result in the spawning of the appropriate windows for input of plate geometry information or for viewing a visualization. Additionally it provides an interface to functionality specific to and contained within a model or visualization environment. An example is the initiation of the calculation of potential values once a geometry has been specified. This functionality is contained within the model environment but access to the functionality is defined in the System Component. The revised system design specifies that this interface is to be implemented in the model environment (or visualization environment where appropriate).

In its current state, the System Component is represented by the class hierarchy in Figure 4.2

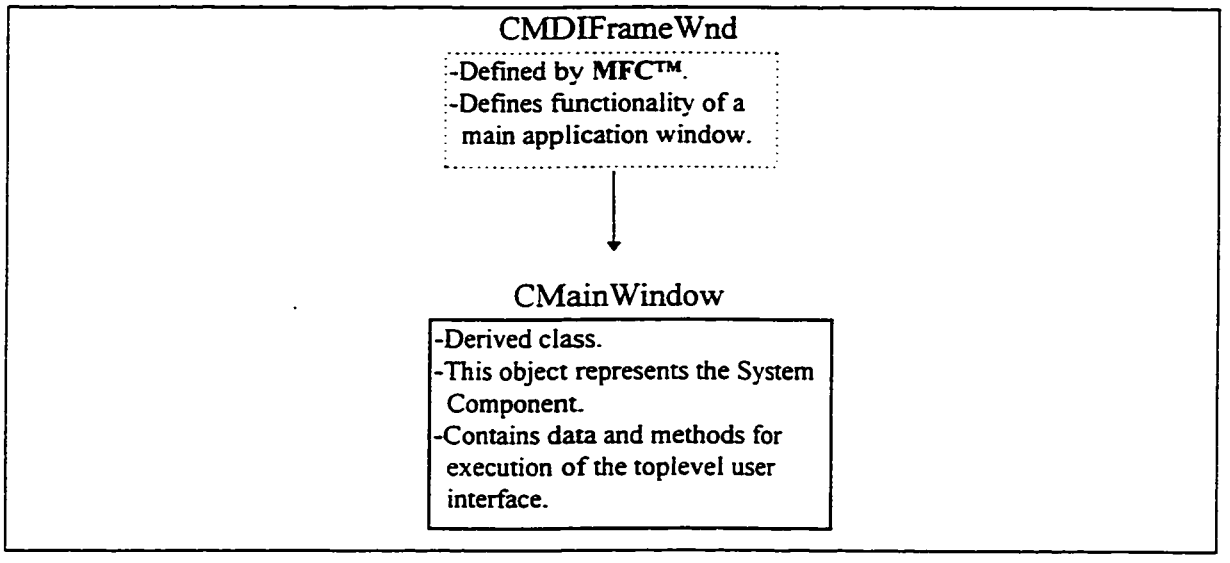

Figure 4.2: The CMainWindow object in this diagram represents the System Component as it currently exists in the implementation. It allows access to existing models, visualizations and various other functionality.

4.5 The Libraries

The discussion of the class libraries, given in this section, constitutes the contribution of this thesis (with regards to implementation) to computer modelling. As a partial implementation only a portion of the functionality and abstraction identified in Chapter 3 is presented.

The libraries perform two functions. The first is the compartmentalization and abstraction of modelling system abstraction and functionality and the second is that they provide the functional set of building blocks from which the **VPMS** system is created.

4.5.1 Visualization Class Library

Presently, the Visualization Class Library encapsulates limited functionality specifically for the creation of visualization environments.

The functionality for the transformation of numerical data into graphical images is implemented and distributed among two visualization environments, which produce a Color Plot and an Arrow Plot.

These visualizations produce visual interpretations of test data generated from a random function which determines values (which represent potential values) at specific points on a grid overlaying a plate geometry. This function is currently substituting for a numerical model and will be replaced when the appropriate model is available.

The two environments are represented by the ColorPlot_Class and the ArrowPlot_Class respectively. Each of these classes have been derived from the Potential_Plate_Class as shown in Figure 4.3.

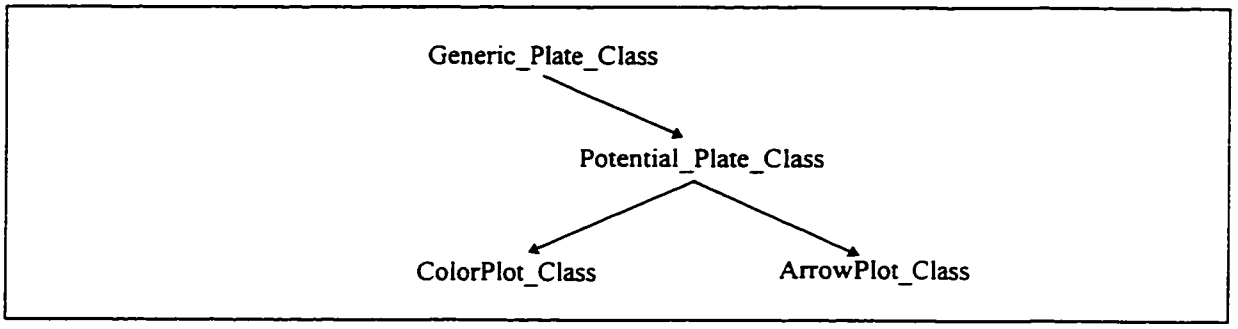

Figure 4.3: Positional representation of the visualization environments.

Also implemented in each of these classes (environments) is limited functionality for communicating with a model environment. For example, the retrieval of data to be visualized as well as data pertaining to the plate geometry which is rendered along with the visualization.

For the purpose of prototyping both the ColorPlot_Class and the ArrowPlot_Class, the functionality responsible for producing the plots exists as member functions of either of the two classes. It is necessary to remove the "monolithically"

implemented visualization functionality within each of the classes and abstract it into relevant (generic) classes which exist as part of the entirety of the Visualization Class Library. An example of this is the functionality for calculating the area of a polygon (a member function of the ColorPlot_Class). This functionality is generic and should be encapsulated in a class structure which can then be called upon to perform its function.

4.5.2 Model Class Library

Several class hierarchies exist for the creation of model "front ends". that is, user interfaces which receive input for a model. The classes include hierarchies for plate objects, electrode objects and property objects

The library does not provide an object for every possible interface element that may be needed, however, through abstraction it does provide a framework for the creation of new, customized, elements which are integrated with those already in the system.

The following subsections describe classes and objects which are explicitly used for the creation of a model environment (within the electrochemical context).

4.5.2.1 Plate Objects

Plate objects are embodied by the Generic_Plate_Class class hierarchy and are used in the creation of an input front end. The class objects encapsulate the appearance, data, properties and operations necessary to create, edit and display a plate. The class hierarchy is given in Figure 4.4 and a more detailed description of the plate class hierarchy is given in Appendix A and B.

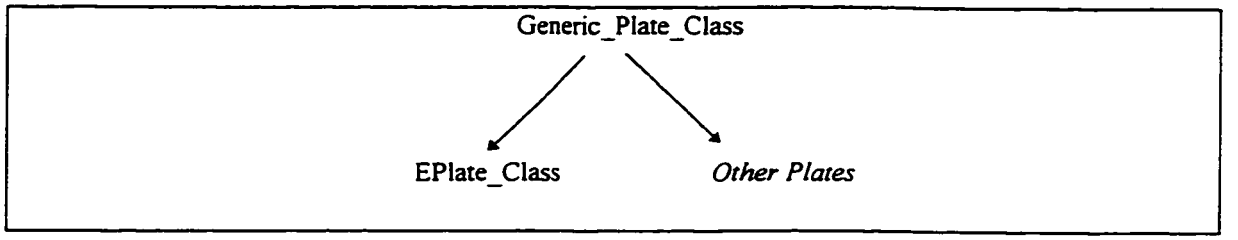

Figure 4.4: The Generic_Plate_Class defines common behavior and attributes while the Eplate_Class represents a specific type of plate. *Other Plates* represent other types of plates not yet created.

Recall that a plate is part of a plate geometry consisting of multiple elements, each with specific properties, about which the electric potential is calculated. A plate element is illustrated in Figure 4.5.

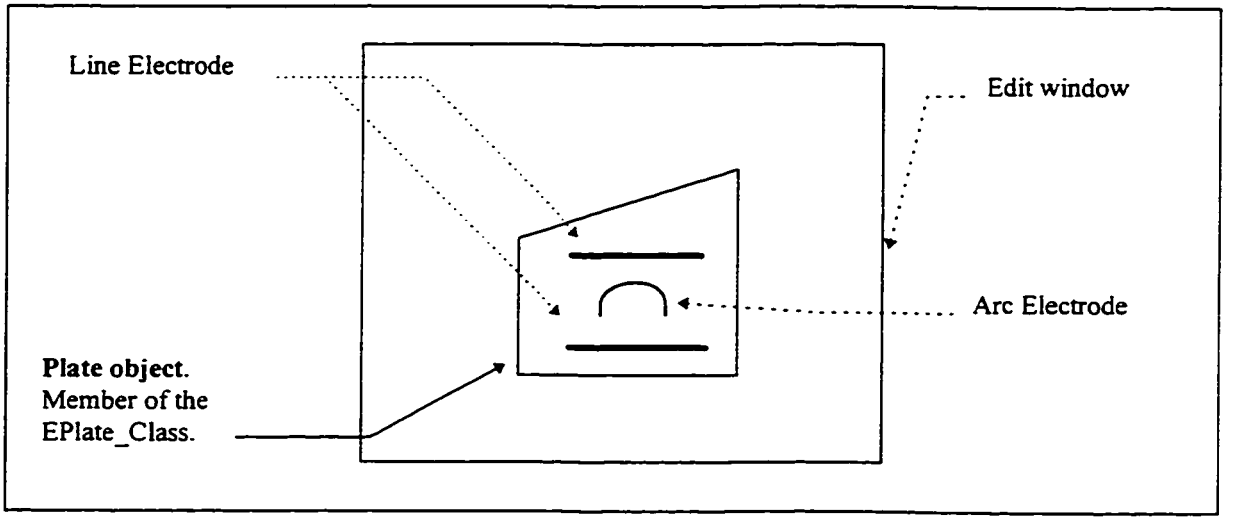

Figure 4.5: A Plate Geometry consisting of a plate object and several electrodes.

There are many different types of plates, both two and three dimensional, although only two dimensional are implemented at this time. With the use of C++, the general data and functionality is abstracted into superclasses from which more specific instances are derived. New plates may be easily created, by programmers, as subclasses. As a result only the specific functionality of the new element need be specified. Much of the functionality of a plate such as editing features and the framework for its integration

and operation is inherited from the superclass. Currently one plate is available within the system and it is depicted in Figure 4.5.

The plate in Figure 4.5, or any others that are added in the future, may not be adequate for the needs of a user. For this reason the system provides the functionality which allows the user to create new ones. Presently, the class hierarchy, rooted at the Generic_Plate_Class, supports this by allowing any pre-defined plate to be graphically modified with respect to size, shape and configuration. This plate may then be save as a new plate.

For a programmer to create a new type of plate, a new plate class, similar to the EPlate_Class, must be derived from the Generic_Plate_Class. The programmer must specify attributes specific to the plate being created, in the new plate class object, such as its shape. Other specific attributes which identify this plate and must be specified include available editing properties and invariants such as no overlapping edges or only 90 degree turns.

Figure 4.6 represents a new plate. This plate may represent a new plate which the user of the modelling system created or it may represent a new plate which was programmed into the system as new plate object derived from the Generic_Plate_Class.

In addition to attributes given to a plate, such as its appearance, there are properties which are used to differentiate it from other plates, or elements, within a plate geometry. These properties are actual class objects which encapsulate physical and behavioral characteristics such as material of construction (of the plate) and its electrical

and conductive properties. These class objects are bound to a plate object to describe and identify it.

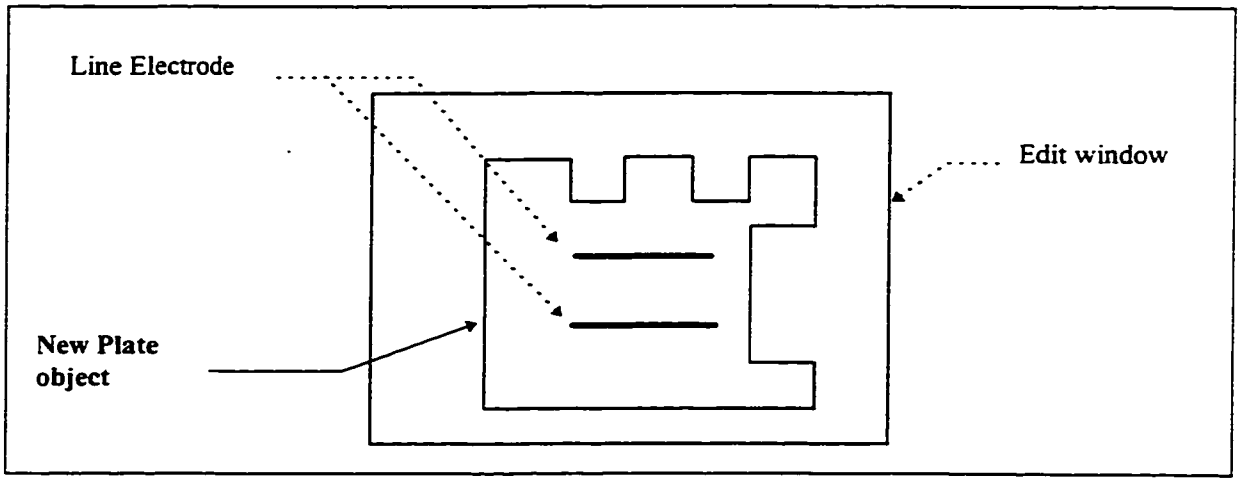

Figure 4.6: A new type of plate object with two charged elements.

4.5.2.2 Electrodes

Electrode objects are embodied by the Electrode_Class class hierarchy. These elements generally have a positive or negative charge associated with them and are used to control or affect the potential values, and therefore the electric field, on and around the plate. Electrodes are displayed in Figure 4.5 and 4.6. The electrode class hierarchy encapsulates the data and functionality of these elements and is depicted in Figure 4.7 below. Refer to Appendix A and B for a more detailed description of the electrode class hierarchies.

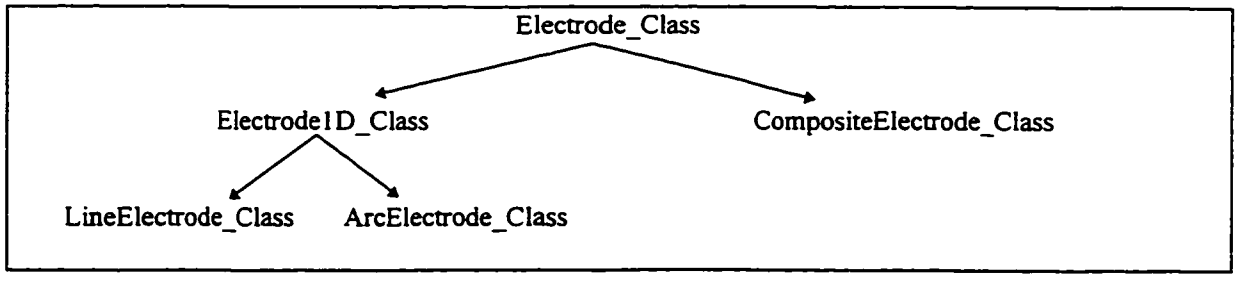

Figure 4.7: The Electrode Class hierarchy.

As in the case of plates it is not possible to have foreknowledge of all required electrodes. Although there are a set of standard pre-defined electrodes, the system also provides a method for users of the modelling system to create electrodes by modifying and/or combining existing ones.

From the perspective of the user, the process used to create new electrodes is the same as that which was used to create new plates from existing ones. One or more electrodes are brought into an editing window and re-sized, reshaped and/or combined as desired. It is then saved as a new electrode. This is illustrated in Figure 4.8.

The CompositeElectrode_Class provides functionality for combining several existing electrode elements. This class keeps a list of all individual electrodes which constitute the composite electrode. Each individual electrode in the composite retains its individual data and functionality. A composite electrode is shown in Figure 4.8

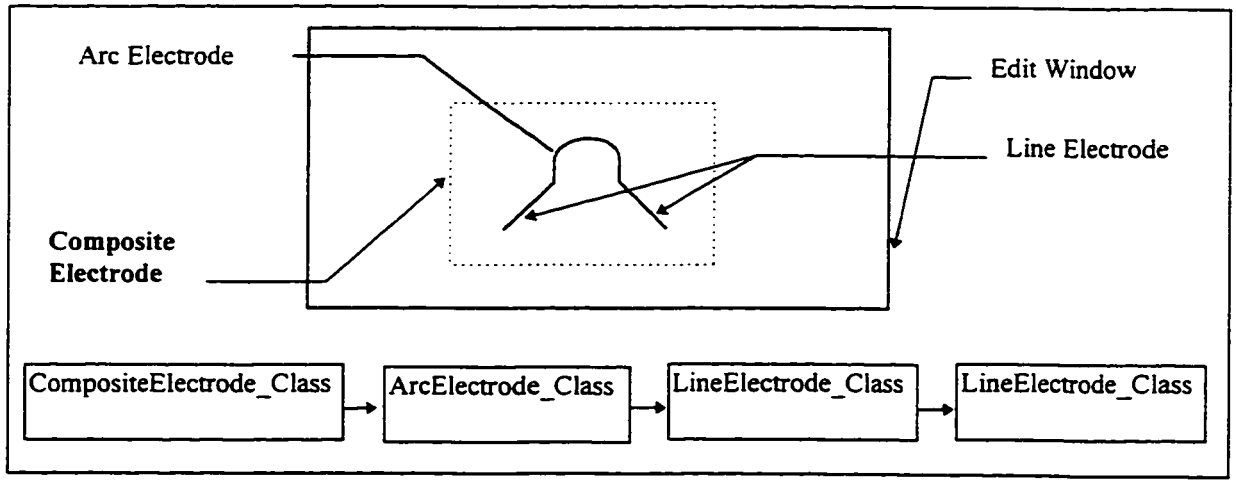

Figure 4.8: The Composite Electrode, is a combination of two Line Electrodes and an Arc Electrode. Each individual Electrode retains its own identity, properties and functionality. The Composite Electrode object maintains properties of the Electrode as a whole as well as maintaining a list of individual electrodes as demonstrated in the bottom portion of the diagram.

A programmer may create new electrodes by deriving new class objects from the Electrode_Class. In addition to the general appearance of an electrode, property objects, functions and data, are required to further define it. Figure 4.9 illustrates a possible electrode defined by a programmer.

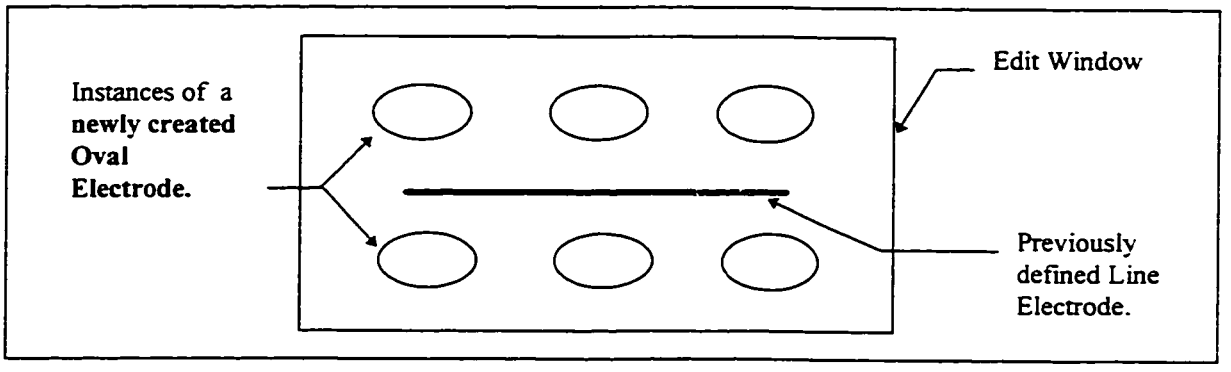

Figure 4.9: New Electrode objects may be derived from the Electrode class. The "Oval" Electrode represents a new type of electrode which contains its own properties, and methods.

4.5.2.3 Properties

Properties are represented by the Property_Class class hierarchy. Creating new electrodes and plates is a matter of creating new graphical objects with a set of attributes or combining and editing existing ones. To further distinguish these objects from others, property objects are used.

Property objects are C++ objects which contain information and methods defining types of properties such as voltage levels, material of construction and boundary conditions for plate geometry elements. Property objects are attached to plate geometry elements such as a plate or electrode.

The types of properties mentioned above represent only a portion of those which are appropriate to electrochemical analysis. The property class hierarchy allows

programmers to derive new property classes which may be required for electrochemical analysis applications or for other scientific domains as well.

The property class hierarchy may be decomposed into two sub hierarchies. The first defines a more application specific set of classes and the second defines more general properties such as material, charge and mass properties and are contained by the former (application specific) classes. The first sub-hierarchy is illustrated below, in Figure 4.10, and is implemented in the system. however, it contains no functionality at this time.

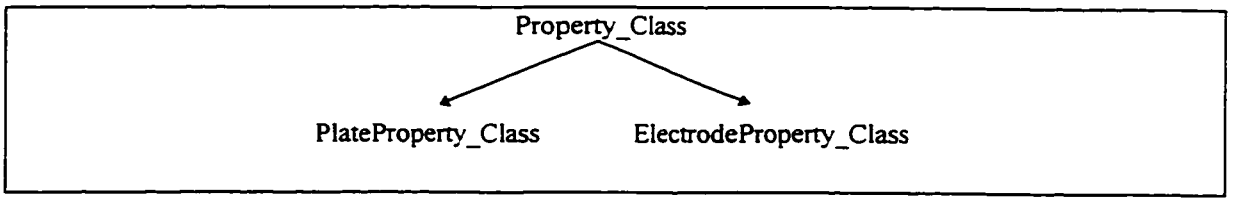

Figure 4.10: The current property class hierarchy.

The use of PlateProperty_Class and ElectrodeProperty_Class provides a mechanism for combining application specific properties and functionality. More precisely, these classes provide a more application specific context for the use of the more general properties. For example the PlateProperty_Class provides functionality of the mass property as it pertains to electrochemistry.

This portion of the class hierarchy changes according to the application. For instance, a magnetic force application will use property objects other than, or in addition to, the PlateProperty_Class and the ElectrodeProperty_Class.

The second portion of the class hierarchy defines more general, application independent properties. These properties include those previously mentioned, such as

mass, charge and material, and may also include any number of others such as capacitance, flux and velocity. These types of properties are useful across different domains and do not include application specific data and functionality. These properties may or may not be encapsulated in more application specific classes.

At the present time, this portion of the class hierarchy is not implemented, however, a proposed hierarchy is given below in Figure 4.11.

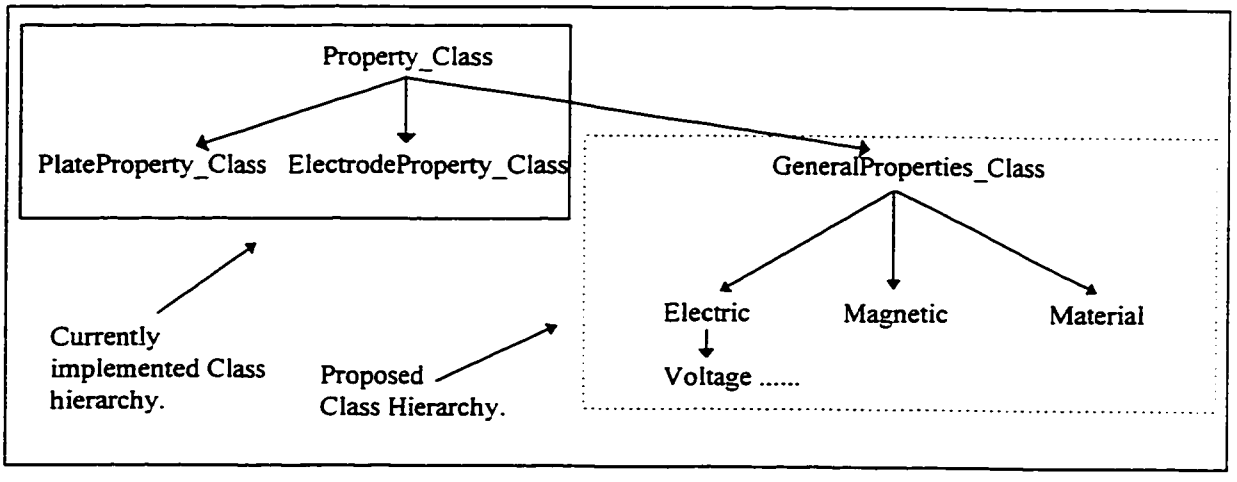

Figure 4.11: Property class hierarchy sub-divided into two. The portion enclosed in the solid box currently exists and is used to isolate property functionality specific to an application. The portion in the dashed box contains a proposed class hierarchy for more general and application independent properties.

4.5.2.4 Comments

This class hierarchy also contains limited data handling capabilities which are used to store the (mock) potential values once calculated.

Because the nature of interface and model creation is not standard, it is difficult to provide a set of standard tools for their implementation. It is, therefore, the goal of this library to provide applications specific tools, such as plates, properties and electrodes, as well as a framework (abstractions) which allows the creation of new tools for use with

other applications. The classes and objects described above represent a small segment and a specific application of the Model Class Library. These classes embody the modelling portion of a modelling system described in Chapter 3.

The interface of the model and the model itself are very closely related because the interface must provide data in an acceptable format to the model. Generally one is created in conjunction with the other, however an interface might serve several related models and vice versa. For example, if the models are similar, as the magnetic force and electrostatic models, then the same interface may suffice because the data requirements and interface metaphor are also similar.

4.5.3 Yoke Class Library

As described in Chapter 3, the function of this library is to handle the details of linking an environment to the System Component (refer to Figure 3.4) as well as providing communication between them. This is provided through abstraction of integration and communication details. In other words this library contains the required objects whose methods may be called to perform tasks such as method invocation or the exchange of parameters, handles, instance values and object references. Placing this abstraction in the Yoke Class frees creators of environments from having to explicitly implement the necessary functionality, they need only invoke appropriate object methods to accomplish a required task. The discussion of the Yoke Class Library assumes that environments are compiled as DLLs.

Note that classes of this library have not yet been created, however, some functionality which belongs in this library exists within the current visualization and model environments. For example, communication functionality (between a visualization environment and the System Component) which requests a reference to a model environment so a plate geometry may be retrieved and displayed in a visualization. Presently, this functionality is implemented as a function call which provides a reference to a model environment. Additionally, the portion of this library which handles initialization and integration details of environments has been implemented (in an **MFC**™ environment) as part of the experimentation on DLLs described earlier. The implementation of this functionality is described below. Note a non-**MFC**™ based implementation is also presented for completeness.

Depending on the operating system and windowing environment used, there will be minor differences in the initialization and initiation of the DLL as well as the coding of the application itself.

To facilitate the initialization and initiation of an environment the Yoke Class Library must provides an entry point into the DLL and from this entry point other Yoke class functions are invoked to begin execution of the visualization or model. Entry points are functions within a DLL which are called from other processes and used to initiate the execution of a DLL. The entry point function need not be a member function of an object. In this instance the entry point function is not an object member function and is defined in the source code files.

To initiate its execution as a DLL, the entry point function within the DLL, *Initialize*(), is invoked by the System Component process as pictured in Figure 4.12. As the name of this function suggests, it performs some initialization and then invokes the execution of the visualization or model. Recall environments are subclasses of Yoke Class classes.

Under a **Win32™** or **Motif™** environment the Yoke Class framework allows the creator of a visualization or model to proceed with development in a manner which is similar to the development of any other **Win32™** or **Motif™** application. The principal difference is that the "*main*()" function (standard **C/C++** and **Motif™**) or the "*WinMain*()" function (standard **C/C++** and **Win32™**) is not used. Instead, the *YokeMain*() function, which is defined in the Yoke Class hierarchy, is used. The *YokeMain*() function is analogous to the "*main*()" or "*WinMain*()" functions. Code which is placed in the *main*() function of an application is placed in the *YokeMain*() function. The *YokeMain*() function is the starting point of a visualization environment or model and is illustrated in Figure 4.12.

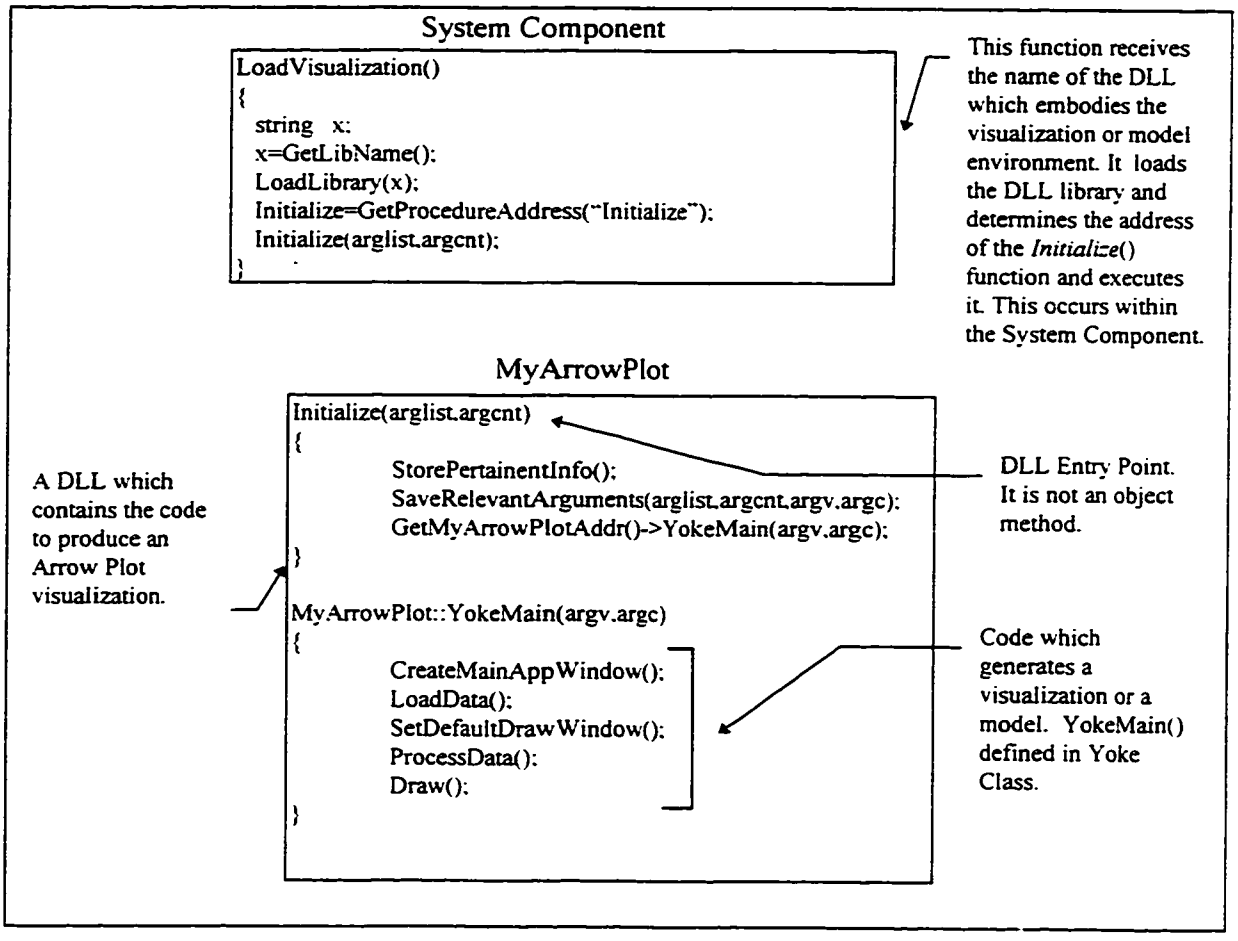

Figure 4.12: The System Component of the modelling system loads the visualization DLL, MyArrowPlot, and invokes the *Initialize*() function. This function performs some initialization and then invokes *YokeMain*(). The type of arguments in arglist could be the process instances, object references window references and depend on development environment. This is not actual code.

In an **MFC™** environment, an application is derived from the, **MFC™** defined, CWinApp class. This class provides functionality to handle initialization, event handling and system interface details. As illustrated in Figure 4.13, a user derives an object from the CWinApp class and from this object the main application window which the user will interact with is created. The derived class, "AnyApplication", is the application object.

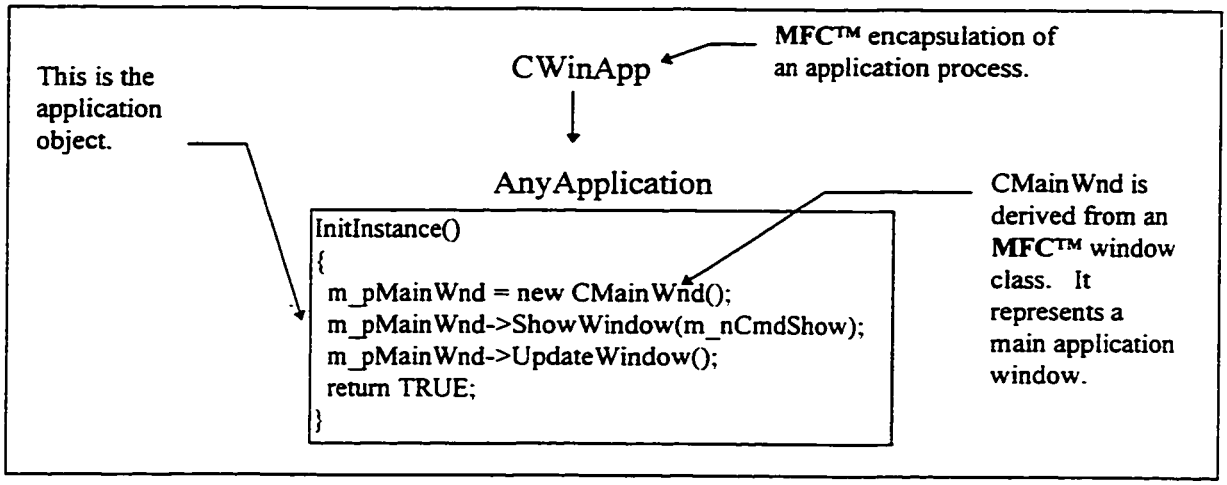

Figure 4.13: The "AnyApplication" object embodies the application which executes on the computer. Within the AnyApplication class the user creates an InitInstance function, which is called by the MFC™ framework, in which the toplevel window/object and interface is created.

Implementing the Yoke Class under **MFC™** requires two changes from **Motif™** or **Win32™** implementations. The first change has the Yoke Class derived from the CWinApp class and the second change is within the *Initialize*() function provided by the Yoke Class. The underlying reason for the first change is because the CWinApp class encapsulates functionality needed by the application (an environment), such as event handling. These functionalities are inherited by the Yoke Class and, therefore, by an environment (through the Yoke Class). The second change is performed because the programmer of the environment does not include a *YokeMain*() function in the application. Instead the InitInstance function is used (as with any other **MFC™** application and illustrated in Figure 4.3). Therefore the *Initialize*() function must invoke the DLL differently.

The result of the second change is that the code within the *Initialize*() function is modified. Essentially the code within this function mimics the code use by **MFC™** to initialize any **MFC™** application, Refer to Figure 4.14. A detail description of the

changes will not be given as any discussion of said changes is not appropriate without a detail discussion of the operation of **MFC™**.

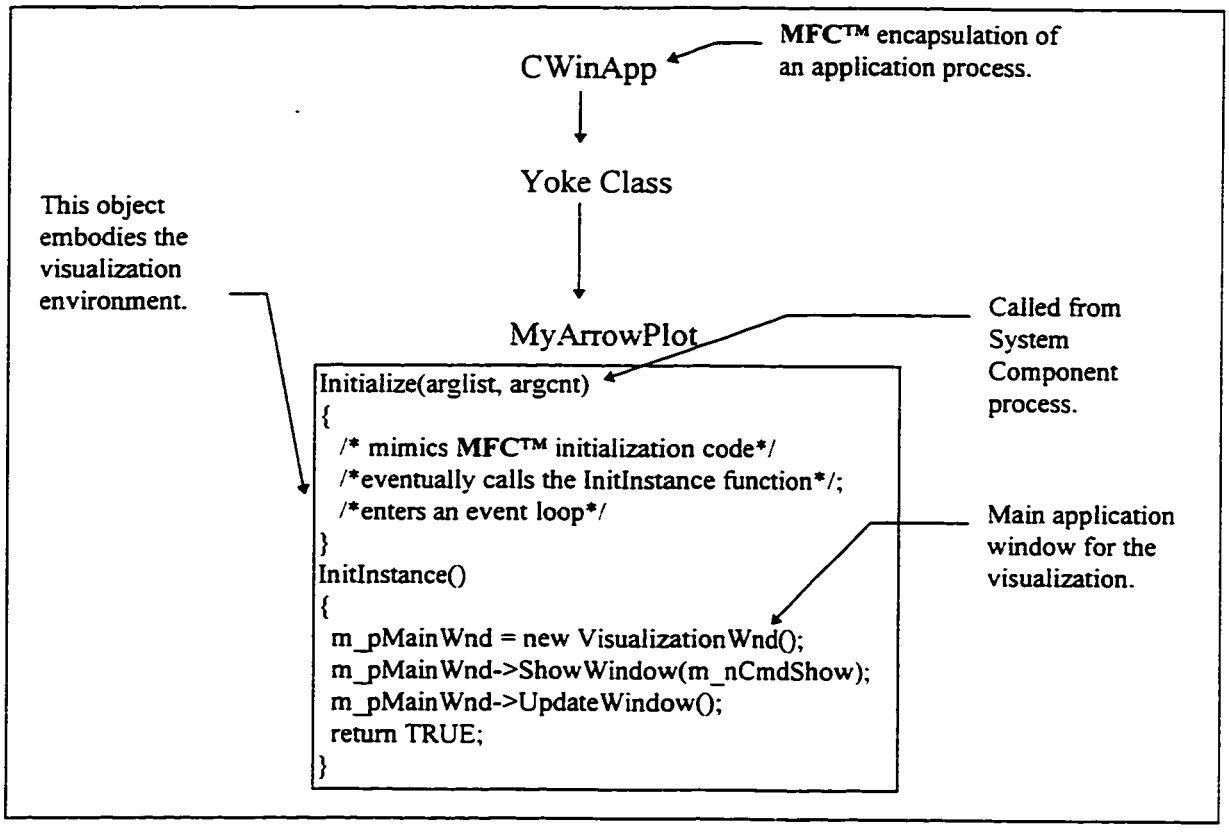

Figure 4.14: In this scenario the MyArrowPlot visualization environment is still derived from the Yoke Class. The difference lies in the *Initialize*() function which essentially uses the code which **MFC™** uses to initialize any ordinary Windows application. Note also that InitInstance() is used instead of the *YokeMain*() function.

DLLs depend on the existence of a currently running process since they are designed to be mapped into the code space of these processes. To enable the DLL to execute as an individual environment outside of the computer modelling system, a small executable program, devoid of any modelling functionality, must accompany it. This executable program provides the running process whose code space the DLL is mapped into. It also provides the command for the execution of a DLL (an environment). The

name of the appropriate DLL can be entered by the end-user and once this information is received the module loads and executes the DLL. This last issue is included for completeness.

4.6 Efficiency

Any program or system which involves computer graphics and/or numerical simulation is inherently computationally intensive. To deal with this complexity and amount of computation, individual modules, i.e. visualization environments or models, must be as efficient in design and implementation as possible. Efficiency is dealt with at several levels of system creation including the design phase and the coding phase.

The issue of efficiency should be considered as early in the development process as possible, that is in the design phase. Where this is not possible through, specification of optimal algorithms and data structures, it is necessary to delay the optimization. Instances where this may occur are with time critical systems which must be completed quickly, or with systems in which there is an incomplete specification of the system. In these cases, [Pressman 92] highlights a strategy to deal with delayed consideration of efficiency. The strategy may be summarized by the following points:

- Develop the system without concern for optimization. Optimization will be dealt with at a later time.
- Use a higher order programming language for development of the system.
- Identify and Isolate segments or modules which are compute intensive.

The above approach is useful and coincides with the prototype approach to the development of systems [Pressman 92] and addresses the issues that affect this system with regards to efficiency. As a prototype the issues of full scale efficiency, (i.e. optimal visualization and graphical algorithms) are delayed. To facilitate improvement in efficiency at a later date, the current design and implementation of the system employs the strategy listed above.

Elements of this strategy can be observed in the **VPMS** system because the models and visualizations are implemented as individual environments. It is possible to optimize the design and execution of each of them at a later date with minimal or no effects to other models, visualizations or system components. For example, it is possible to use a more efficient method of interpolation in the Color Plot with few or no required alterations to other visualizations or system components outside of the Color Plot. A possible modification which might have to be made to other components is the format in which they supply data to the Color Plot.

This design also isolates the areas of computational intensity because it is precisely the visualization environments and the models which account for the majority of the CPU time.

The use of object oriented programming serves to modularize areas of inefficiency, allowing them to be replaced with more optimized modules. For example, objects for mesh generation can be substituted for more efficient ones. If the interface remains the same then any system component or algorithm that references this object to

perform mesh generation need not be concerned that the object has been changed because of the fact that input and output remain the same.

Aside from algorithmic and system design efficiency, it is possible to improve the code efficiency of a system. Steps have been taken in this direction with regards to the implementation of this system. These steps include:

- The simplification of arithmetic and logical expressions.
- The use of faster arithmetic operations to replace slower ones, i.e. substitute addition for multiplication.
- Elimination of multi-dimensional arrays.
- Removal of insignificant statements outside of loops.
- Reduction of repetitive nested de-referencing of objects and structures.
- Removal of function calls from large and nested loops.
- Passing references to large data objects instead of copying data.
- Reduction of the number of objects dynamically constructed and destroyed.

These are examples of the types of programming optimizations which exist throughout the implementation. It suffices to state that not every instance for code optimization has been accounted for and as a prototype there remains much work to be completed. To aid in these types of code optimizations tools exist within compilers which identify and optimize instances of inefficiency. This level of optimization is to be performed on more complete versions of THE SYSTEM.

4.7 Discussion

This chapter discussed issues concerning the use of C++ for implementation and its role as an object oriented language as well as issues of portability with respect to windowing systems and rendering libraries. The implementation of models and

visualization environments as DLLs was then explained followed by a description of each of the three libraries and how they are used to support the creation of visualization and model environments. The issue of efficiency and how it is dealt with within this prototype was then discussed.

Reasons for implementation of the libraries is to provide functionality (both application specific and application neutral) in layers of abstraction which can be aggregated to construct environments and ultimately satisfy the goals of the system design. The libraries provide creators of modelling system functionality with a functional set of tools or building blocks.

Presently, the functionality in each of the libraries is limited because it is in the prototyping stages of development. As the libraries grow and are refined the modelling system will become more functional and eventually embody the design as described in the previous chapter.

Chapter 5 FUTURE WORK

5.1 Introduction

As previously stated the initial design and implementation of this system, detailed in Appendix A, is a prototype. Examination of this prototype yielded a new design which has been discussed in Chapter 3. Much of the initial implementation supports the new design, however several modifications are required. These are discussed below, in Sections 5.2, 5.3.1, in the context of the class libraries and of the System Component.

The methods by which this design deals with modularity and distributivity leave it open to future research and implementation possibilities with **Java™**. Future work in this direction is discussed in Section 5.3.2.

Future work is also possible with respect to the ORM model [Bronts 95] [Creasy 96] which defines a theoretical kernel for object role modelling, however this is not discussed herein.

5.2 Immediate modifications

Several immediate modifications are necessary to update the current implementation of the system so that it more accurately represents the design. The required modifications are to each of the three libraries and are given below.

5.2.1 Visualization Class Library

Currently, the ColorPlot_Class and the ArrowPlot_Class classes have been derived from the Potential_Plate_Class as seen in Appendix A. It is required that these classes be moved to another position in the class hierarchy. These classes should be derived from the Yoke Class, and its hierarchy, as shown in Appendix B, Figure B1. This movement will facilitate the development of visualizations, and models, as individual environments as per the design outlined in Chapter 3.

For purposes of prototyping, each of the Color Plot and the Arrow Plot visualizations have been implemented such that all functionality related to each of the plots has been programmed into the ColorPlot_Class and ArrowPlot_Class classes. Another immediate modification which is necessary to each of these visualizations is the removal of some of the "monolithically" implemented visualization functionality, such as the reading and manipulation of data.

It is also required that the visualization environments, as well as the models be compiled as DLLs. Presently, they are integrated and compiled as a part of the system.

5.2.2 Model Class Library

Currently, classes in the Model Class Library support the creation of plate and electrode objects. Although the application specific classes for properties, such as the PlateProperty_Class and ElectrodeProperty_Class exist, they do not possess any functionality. Any properties which are required by geometry elements, a plate for example, are contained within that specific element's class. It is therefore necessary to

remove any of these properties, along with the relevant functionality, and implement them in the appropriate property class.

It is also necessary to create the class hierarchies for the more general properties, as discussed in Chapter 4. These are properties such as voltage, temperature, material of composition, for example. They specify general properties which can be used across application domains and are used in conjunction with the more application specific properties such as the PlateProperty_Class and the ElectrodeProperty_Class.

The model environments allow users to create new plates and electrodes through editing and modifying existing ones, such as combining multiple electrodes to create a new composite electrode. It is necessary to implement the functionality which allows a user of the system to create entirely new elements; that is, not created by editing a pre-existing geometry element. This requires a graphical editing environment which provides the user with a basic set of tools, such as lines, squares, etc., and allows them to create and save new elements.

5.2.3 Yoke Class Library

Visualizations and models are currently integrated within the system and each contains the functionality which allows them to communicate with the system, i.e. the passing of data or window handles. It is necessary to remove this functionality and create abstractions which belong in the Yoke Class Library. This allows classes within this library to contain functionality necessary for integration of environments with the modelling system. Each environment must then be compiled as a DLL.

Additionally, the functionality for initialization and initiation of environments (recall the experimentation with DLLs) must be placed in the Yoke Class Library.

5.3 Java

The system design provides an additional method for distributing modelling capabilities through the use of "**Java™**-like" functionality. A description of this method as well as possible future directions are given.

5.3.1 Incorporation of Java™

The system design provides for a distributed modelling capabilities through the use of functionality provided by a language such as **Java™**. As previously discussed, in Section 3.3.2.3, the design allows remote users to retrieve an interface to a model, or a visualization. The interface which allows input of parameters, i.e. a plate geometry, is to be implemented as a **Java™** *applet* which allows the user to interactively enter the desired parameters to a model or visualization. The input information is then returned to the system and made available to the appropriate model or visualization.

A limited prototype has be created which defines *applets* (analogous to the interface *applets* referred to above) which can be retrieved over a network and accept input of parameters. This data may then be transferred back to the server [Martincic 97].

Presently the implementation allows a connection to be accepted from a remote machine. Once a connection is accepted the modelling system can return an HTML document. Modifications to the implementation required to complete this **Java™**

functionality include the abstraction into one or more classes of the code which accepts multiple network connections so that more than one request can be processed at a time.

It is also necessary to modify and incorporate the prototype of [Martincic 97] into the implementation. This will provide the system with the appropriate interface *applets* and the functionality for the transfer of data between the modelling system and the *applets* executing remotely.

5.3.2 Future Implementation of Java™

With the increasing number of extensions to the **Java** language it may soon be possible to create individual visualization environments and models entirely in **Java**. If this is the case, then these visualizations and models can be retrieved from any site and executed within a web browser without the risk of them causing damage to the local machine.

To enable system environments to be implemented as **Java** *applets* which execute in a web browser, these *applets* must have a method of accessing necessary data. There are two possible ways in which this might occur.

The first method requires that the *applet* communicate, via a network connection, to the server from which the *applet* was retrieved and access the data. As data transmission speeds increase, the problems of transmitting large amounts of data, such as long waiting periods and interrupted transmission, become less of a concern. This type of interaction, however, is restricted because the *applet* can only retrieve data from the server from which the *applet* was obtained.

A second approach allows the *applets* to communicate with other "trusted" *applets* or **Java™** applications on remote machines. This makes it possible to retrieve data from any server. This is a functionality which will be included with future versions of the **Java Development Kit™**. [http://www.javasoft.com/]

To increase the versatility of this approach, it should be possible for a web browser based *applet* to retrieve data on the local machine. Instead of connecting to trusted *applets* on a remote machine, it is possible to create a network connection with a **Java™** *applet* or application running locally on the same machine. If the *applet* running in the web browser can trust a **Java™** application executing external to a browser (on the same machine) then a connection for data communication can be made. The trusted application can serve as an intermediary between the local disk and the *applet* running in the web browser by passing data back and forth. The same principle works for a model created as an *applet*.

One problem with this scenario is the availability of an adequate rendering library written in **Java™** for use with **Java™** applications. There do exist several libraries for 3 Dimensional graphics but the extent and effectiveness of their operation is unknown at this time. A possible solution is the use of VRML, and abstractions for the automatic creation of VRML files, to render visualizations and provide interactive access to them [Wood 96].

Until such time that **Java™** rendering libraries, comparable to **OpenGL™** and the like, area available two options are possible. The first option requires that each visualization environment is wholly responsible for its own rendering using any available

drawing tools or classes. The second option takes advantage of **Java™** native methods and currently available rendering libraries such as **OpenGL™**. Using native methods, visualization environments written in **Java™** can access compiled code, such as **C**, in order to access existing rendering libraries. This method, however, once again raises the issue of security.

Future research and design issues for **VPMS** also may include the creation of new visualization environments and models in the **Java™** programming language using the second of the two options just mentioned above. These models and visualizations will coexist with existing visualizations already implemented in **C++** and operate within the System Component, which is also a **C++** implementation.

Communication between the system, models and visualization environments would take place through compiled code which the visualization environments and models can access through native methods. This type of design would require the Visualization, Model and Yoke Class Libraries to be written in **Java™** and would access the compiled code, via native methods, to communicate with the system.

Undoubtedly the **Java™** programming language will be an important component in the construction of future distributed computer modelling systems.

Additional possibilities regarding the combination of languages is the use of languages such as **TCL** which can be used to "wrap" functionality, at various levels of abstraction. Such an encapsulation provides access to functionality combined with the advantages of an interpreted language such as acceleration of the development process.

Chapter 6 CONCLUSIONS

6.1 Discussion

The context of this research is the development of a computer modelling system which scientists and engineers find useful in research and development within a variety of application domains.

Based on a set of initial criteria and requirements, a design for a modelling system, specifically for electrochemical research, was devised and a partial implementation created as a prototype. Additional specifications of both functional and non-functional system requirements and goals prompted a revision of the original design which satisfies the additional goals and requirements as well as previous ones.

Development of a modelling system requires a substantial information. As a result of research conducted in modelling, scientific visualization, software engineering and the field of electrochemistry, valuable information was obtained which provides the foundation for the design and implementation of this system. Relevant information includes:

- Intended users of the system, engineers vs. programmers.
- Requirements of users in a particular research domain.
- Type of interaction required.
- Current modelling systems and their design, intended users, nature of interaction, strengths and weaknesses.
- Current scientific visualization systems and their designs, intended users, nature of interaction, strengths and weaknesses.

This information is used for the specification of the modified design which can be extended beyond electrochemistry for use in other application domains.

6.2 Satisfaction of Goals

As stated in the introduction, the goal of this thesis was the development of a computer modelling system which has its major components designed as individual, self-contained environments. The result is a modular, extensible, open, distributable, modelling system which can access a large number of visualizations.

The revised design addresses these issues through identification and compartmentalization of modelling and visualization functionality. The model and visualization environments embody this seperation and each encapsulates specific layers of modelling system functionality. For example, the visualization environment contains layers of abstraction which provide for the creation of visualizations. **VTK** is an example of a layer of visualization abstraction which is used solely for the creation of graphical images.

The system supports modularity since an environment (model or visualization) can be provided to other users who can incorporate them into their versions of this system. For example old environments can be replaced with updated versions, as opposed to acquiring an entirely new release of the system to obtain the added functionality.

The extensibility of the system is provided by these environments as well as the class libraries which are used to create them. The system can be extended, in a specific

application domain, by adding new model or visualization environments. Electrochemistry has provided a context within which the underlying system development philosophy was tested. System architecture and design was tested through the development of prototypical functionalities contained in the class libraries. Alternatively, the system may be extended through addition of software layers to the libraries, such as creating new plate or property classes. The system may be extended by its creators or by programming knowledgeable users.

The system is open because through the use of the class libraries, anyone can create or modify the environments thus building up a repository of application specific, as well as, more general modelling and visualization functionality.

The system is distributable since models and visualization environments are self-contained; thus, they can be moved from one system to another of differing version specifics. The environments also support the ability to execute outside of the whole modelling system. In other words they may be obtained and executed without the need for the entire modelling system; environments may be executed autonomously.

The design supports access to a large number of visualizations and models. Any number of environments capable of integration with the system can be created independently and stored on a machine. When desired, a visualization or model is executed.

The system supports reuse, at a component level, because visualization and model environments can be distributed to other versions of the **VPMS** system. This may be to extend the system by adding new environments or to provide updated versions of an

environment. It is possible for visualizations to cross application domains. For example, an arrow plot can be used in conjunction with a magnetic force model as well as an electrostatic model as long as input data (into the visualization) is consistent.

The structural design (through implementation of added abstraction) is capable of supporting necessary tools which its users may require. These include:

- Interactive numerical models for simulation and appropriate interface methods, such as the graphical editing of a plate geometry.
- Interactive visualization capabilities for interpretation of results.
- Functions for secondary computation. For example, calculation of **E** (electric field) from the electric potential.

The implementation, in its current state is incomplete (it does not provide all of the functional and non-functional requirements of Chapter 3) and represents a prototype of the system and its class libraries. More work is required to completely realize the design of the system through added levels of abstraction and functionality.

As mentioned in Chapter 3 the nature of distributivity of environments does not support any security. As a result it is possible for a malicious model or visualization environment to damage a system.

Additionally, because environments encapsulate all required functionality for their operation, similar environments will contain duplicate code. Therefore, the execution of two or more environments will result in increased space overhead. For example, multiple copies of the same object may exist at one time.

Another consequence of the encapsulation of functionality in environments is that creating them is a larger task because the programmer must explicitly specify actions that might otherwise be abstracted into another system layer. For example, each environment

must contain its own user interface as opposed to a standard interface which works for all environments.

Although these qualities are undesirable, they are necessary concessions which must be made to achieve the system design and its goals.

One measure of the usefulness of the **VPMS** system and its design will be determined by its acceptance by those for whom it was intended. Another measure is the ease with which the system can be extended (within electrochemistry or other domains) to meet future research and development needs.

Appendix A: Current Class Hierarchies

This appendix contains the class hierarchies of the initial prototype design and implementation. These hierarchies are illustrated in Figures A1 to A6. The design described in Chapter 3 includes several modifications to these class structures. These modifications are illustrated in appendix B.

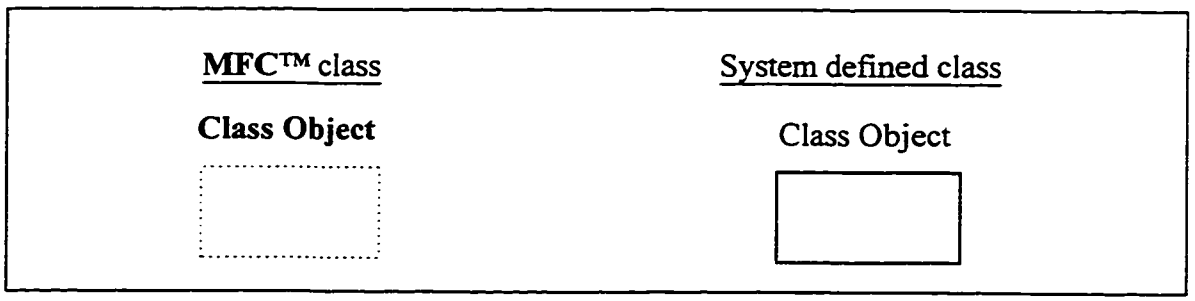

Figure A0: KEY. Class objects represented by the dashed square and bold text are defined by MFC™. Class objects represented by a solid square and normal text are defined by the implementation of this system.

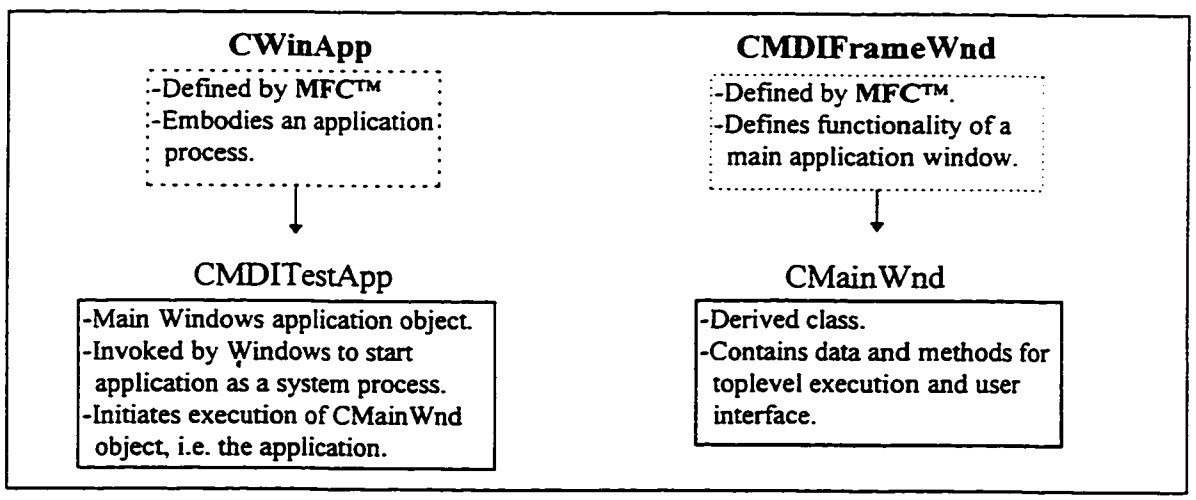

Figure A1: In the class hierarchy on the left, the CMDITestApp class is derived from the MFC™ defined class CWinApp. The CMDITestApp class object is created by the system when the application, modelling system, is started. This object embodies (encapsulates) the application. Within the CMDITestApp object an object of CMainWnd is created. In the hierarchy on the right, the CMainWnd class is derived from the MFC™ defined class CMDIFrameWnd. The CMainWnd class object represents the top level window which will be displayed when the application is started. It is created in the CMDITestApp object.

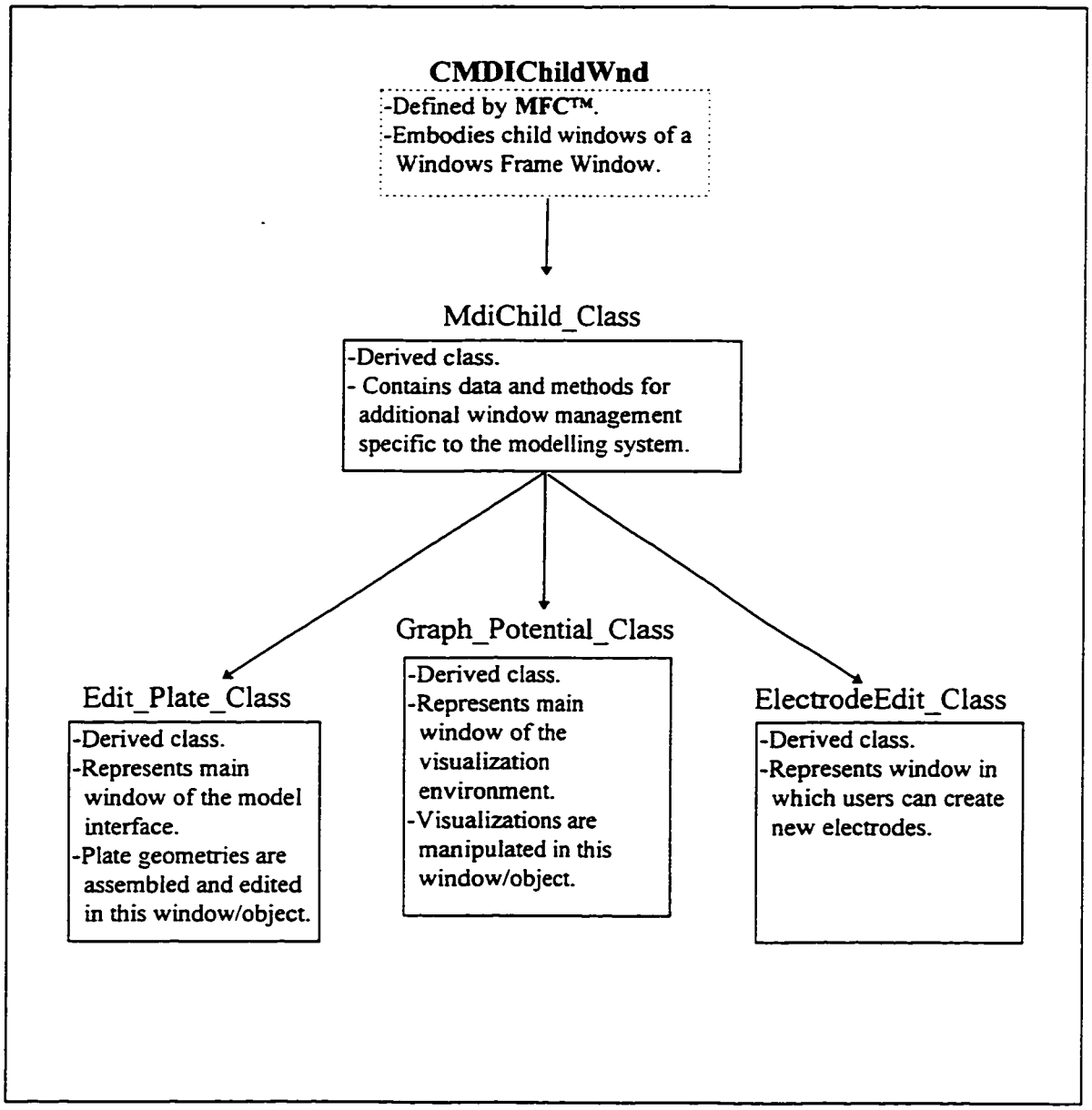

Figure A2: The bottom three classes, of this hierarchy Edit_Plate_Class, Graph_Potential_Class and Electrode_Edit_Class, are window classes. They are derived from the MdiChild_Class which is used to define any application specific functionality that is required by the three windows. This class is derived from an MFC™ class which abstracts the window handling functionality. The Edit_Plate_Class represents the window in which the plate geometry is constructed. The Graph_Potential_Class represents the window in which a visualization is displayed, i.e. a color plot. The ElectrodeEdit_Class represents the window in which electrodes can be edited to create new ones.

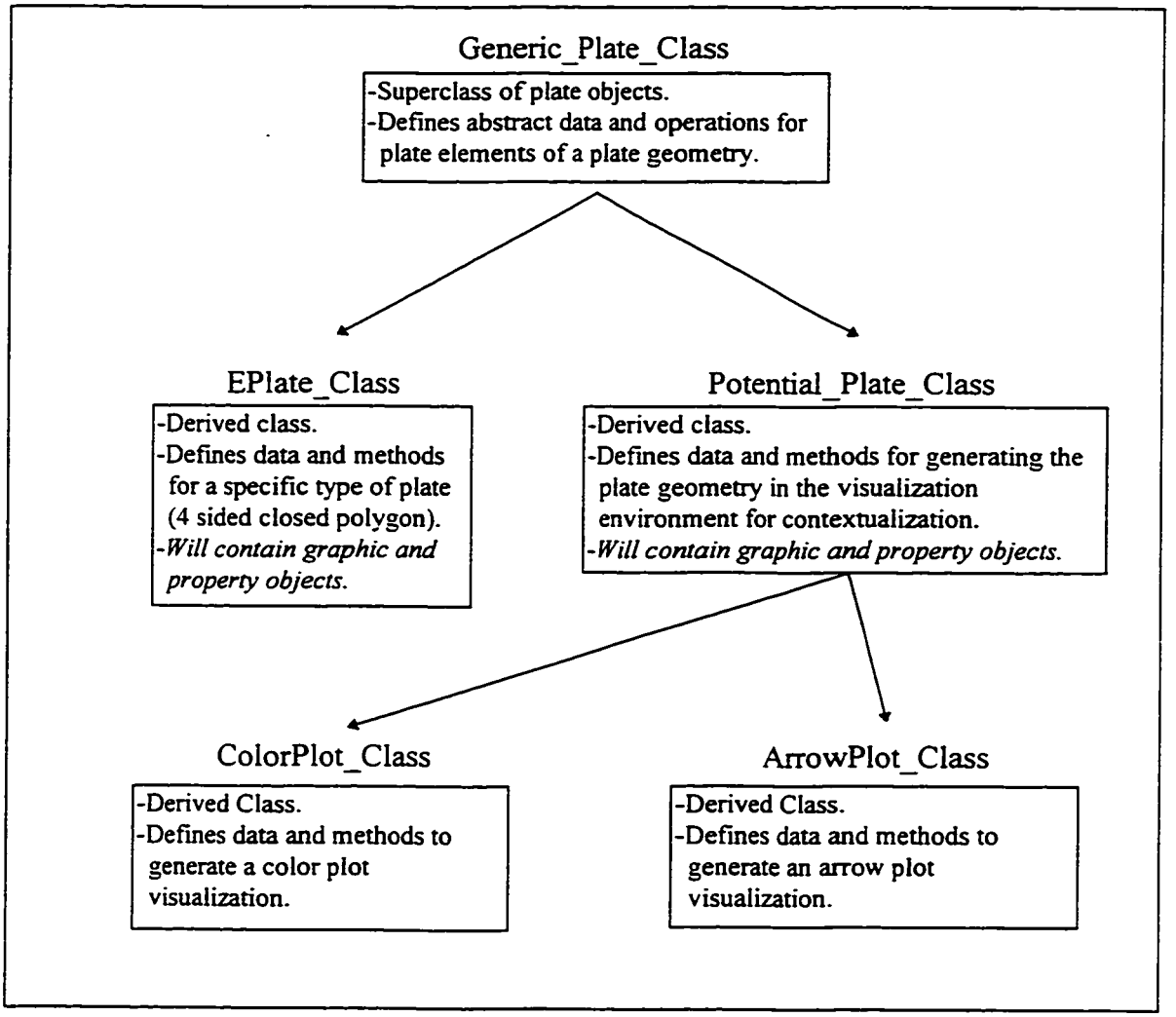

Figure A3: The Generic_Plate_Class defines functionality which is general to the plate elements of a plate geometry. From this class the EPlate_Class is derived and contains the functionality and attributes specific to this type of plate. Also derived from the Generic_Plate_Class is the Potential_Plate_Class. This class contains information relating to the plate objects. It is used to provide access to the plate geometry information to the visualizations. This is achieved through inheritance as both the ColorPlot_Class and the ArrowPlot_Class are derived from the Potential_Plate_Class. Plate geometry information is inherited by each visualization and used to draw the plate geometry in the visualization to provide a context for the interpretation of data.

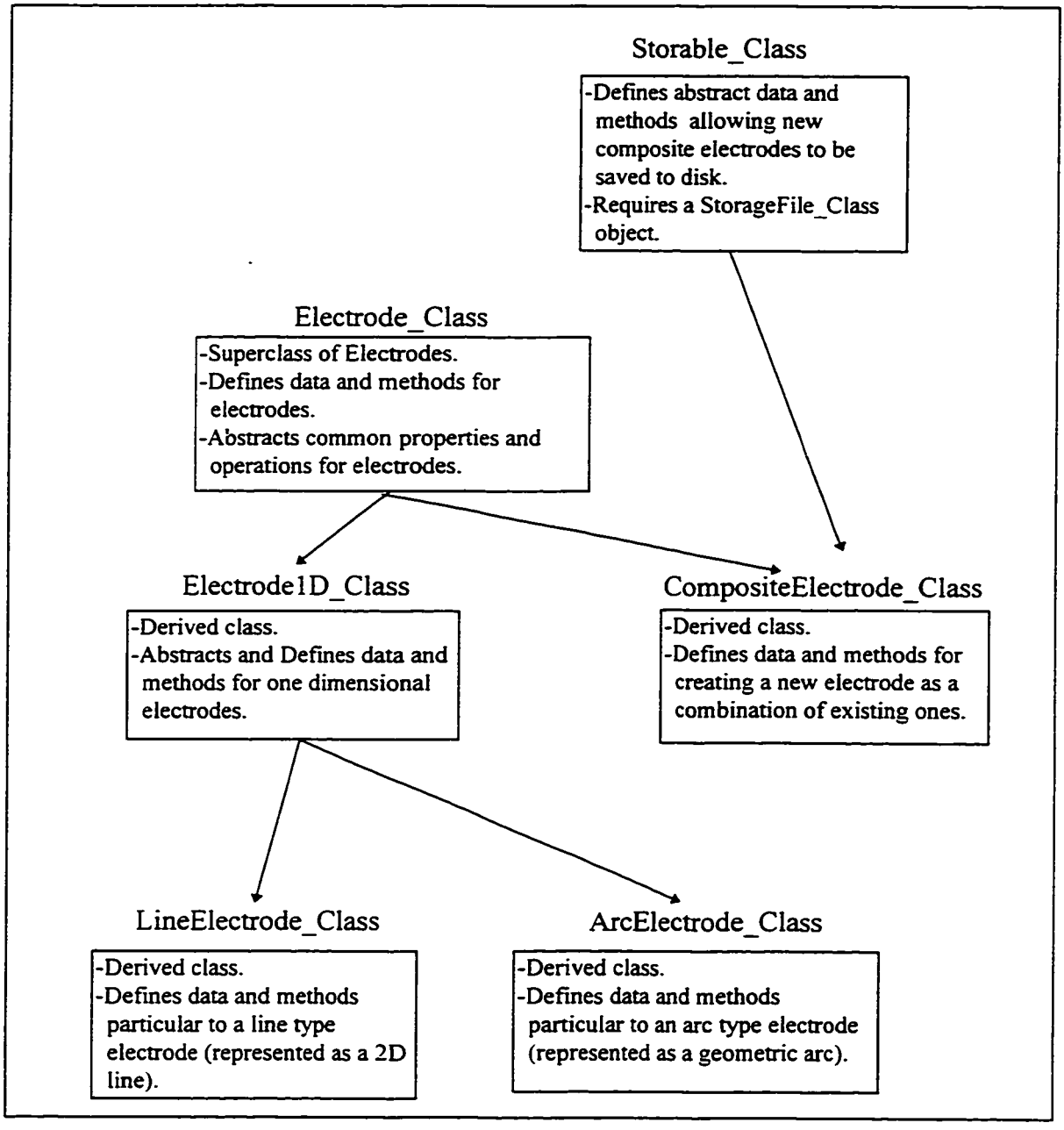

Figure A4: This class hierarchy is used for the creation of electrode elements. The hierarchy is rooted at the Electrode_Class which defines general information and functionality for all electrodes. From this class the Electrode1D_Class is derived along with the CompositeElectrode_Class. The Electrode1D_Class contains data and functionality which is pertinent to electrodes which are one dimensional in nature. Additional classes would be created to handle two and three dimensional electrodes. CompositeElectrode_Class provides necessary functionality to allow a user to create a new electrode by combining pre-existing ones. It maintains a list of individual electrodes which compose a composite electrode. This class also inherits from the Storable_Class. This class provides functionality enabling the new composite electrode to save itself. The two specific electrodes, LineElectrode_Class and ArcElectrode_Class are instances of a one dimensional electrode. These classes are responsible for the specific behaviors of the arc and line electrodes.

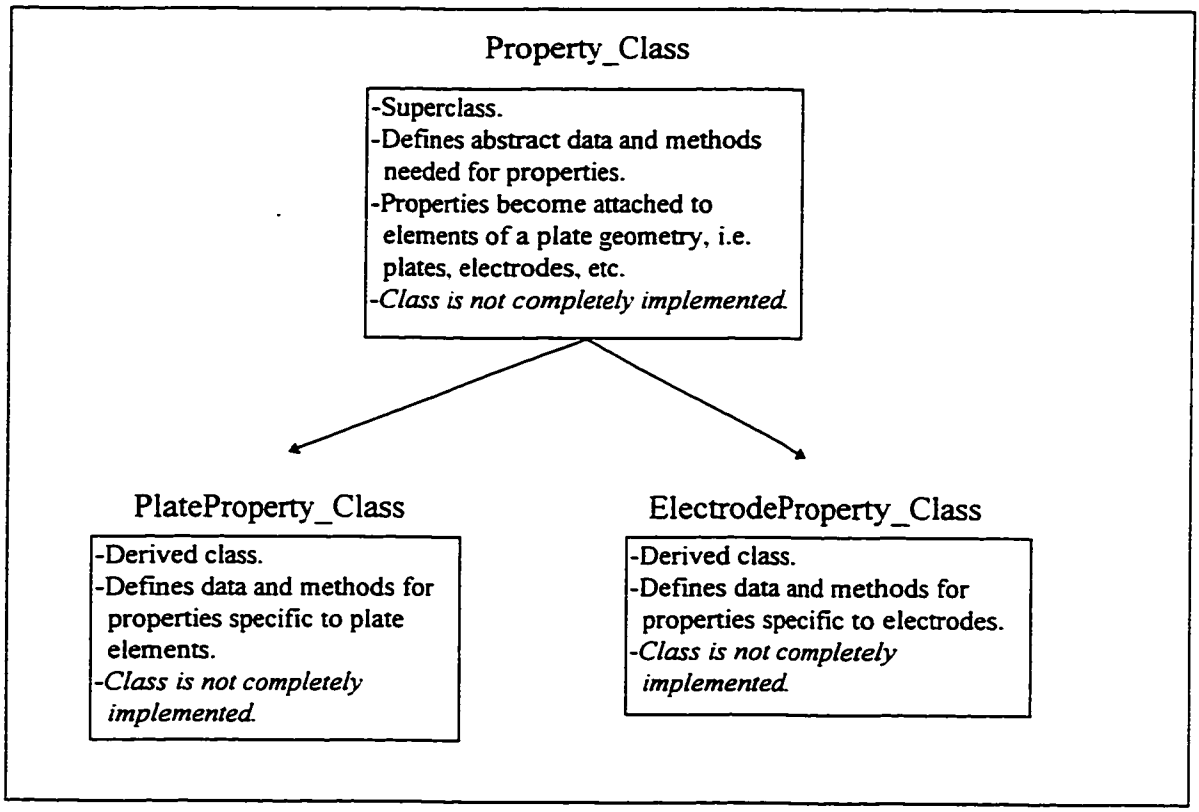

Figure A5: This hierarchy embodies the properties which are attached to plate geometry elements. This hierarchy is specific to the electrochemical application domain. ElectrodeProperty_Class contains properties which are specific to electrodes as they are used in this application. The same is true for the PlateProperty_Class. These classes provide a wrapper for more general properties such as voltage, heat, material of composition, and many others. These properties are not specific to any application domain and therefore do not possess any application specific functionality.

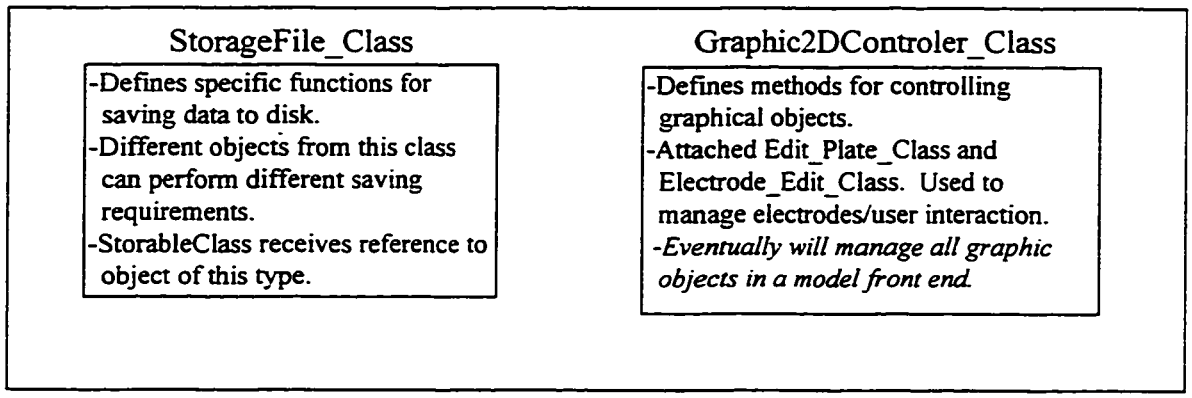

Figure A6: The Graphic2DControler_Class is a class which is used to manage the plate geometry object. This class is attached to the EPlate_Class and keeps a list of all the elements in the plate geometry such as the electrode objects. When an electrode is selected by the user, for editing, a message is sent to the Graphic2DControler_Class which identifies the element that was picked and sends a message to that element which then performs the desired editing operation. The StorageFile_Class is used in conjunction with the Storable_Class and defines functionality for writing information to disk. For example, information within an a class which encapsulates electrode functionality.

Appendix B: Modified Class Hierarchies

The design of Chapter 3 is a modified design which resulted from an examination and improvement of the implementation of the initial system prototype presented in Appendix A. The hierarchies in this appendix represent the modified system design as described in Chapter 3. The differences between the class hierarchies from appendix A and B are listed below followed by a pictorial representation.

- Creation of the Yoke_Class. Note that functionality of the Yoke_Class is distributed among several different classes (ColorPlot_Class, Edit_Plate_Class) in the original design.
- Visualizations environments, i.e. ColorPlot_Class and ArrowPlot_Class are now a subclass of the Yoke_Class.
- Models, ElectrostaticModel_Class, are a subclass of the Yoke_Class.
- User interface windows are created and handled within ColorPlot_Class, ElectrostaticModel_Class and the ArrowPlot_Class.
- The PotentialPlate_Class has been removed from the Generic_Plate_Class hierarchy.
- The **MFC™** window from which the MdiChild_Classes is derived from is a CFrameWnd and not a CMDIChildWnd as in the previous implementation.
- Note: Objects of type Generic_Plate and Electrode_Class will be given to visualizations to enable them to render a plate geometry, or part thereof, so that the actual visualizations, such as arrow plots, can be superimposed upon them giving the resultant visualization an interpretive context.
- Note: Not all classes are represented herein. Classes for interface objects such as buttons and dialog boxes are not displayed since they do not add any meaning to the class structures in this appendix.

The class hierarchies and structures of the modified system design are illustrated in Figures B1 to B7. Refer to the key in Figure A0.

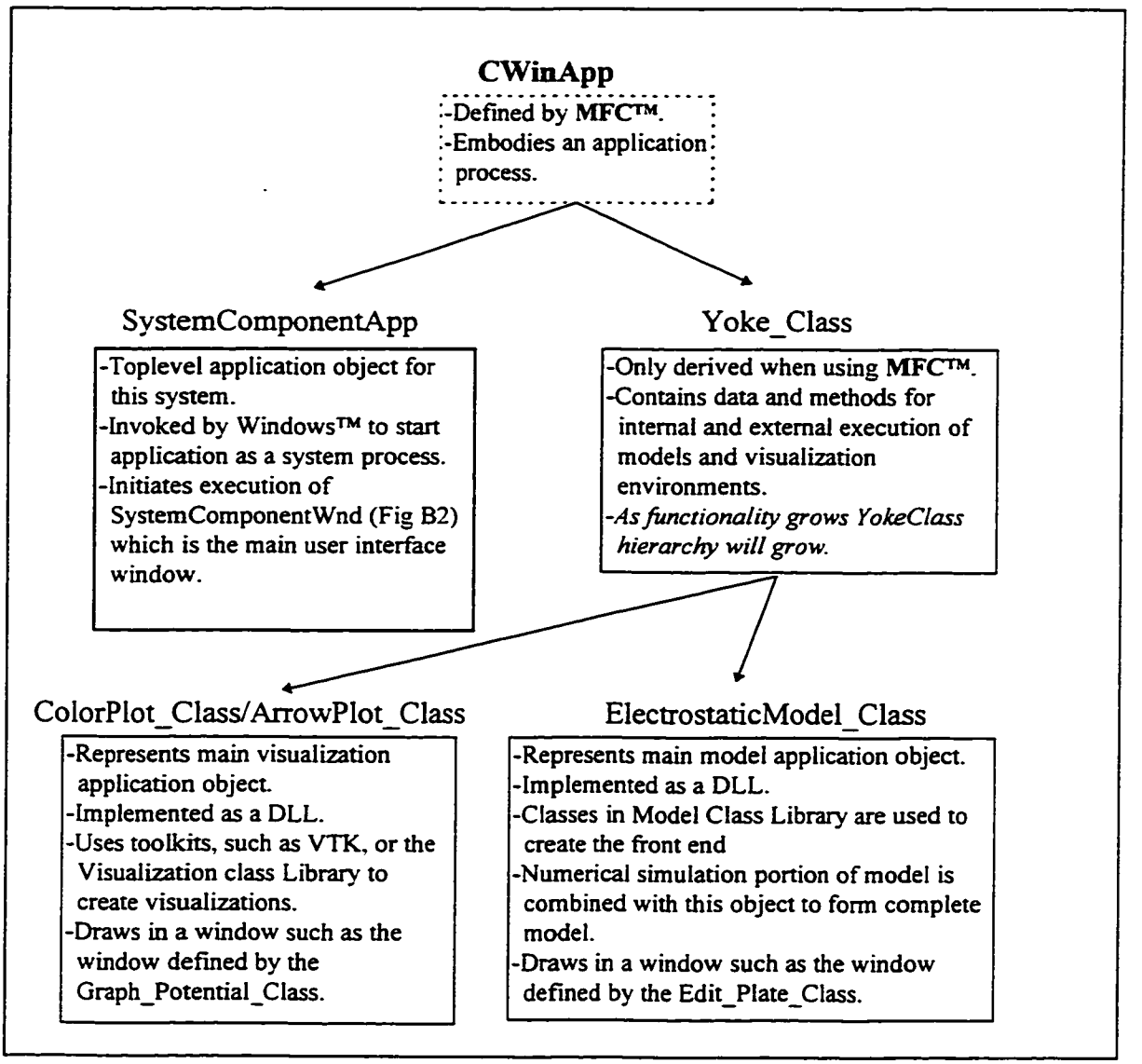

Figure B1: CWinApp is an **MFC™** defined class which embodies an **MFC™** application. In this case the Yoke_Class and the SystemComponentApp are derived from CWinApp. The SystemComponentApp is similar to the CMDITestApp of Figure A1 and represents the main application object for this system (the System Component). This class contains the toplevel user interface (part of the System Component, depicted in Figure B2) and provides a point of departure for accessing visualization and numerical modelling functionality. The Yoke_Class is also derived from the CWinApp class so that it can provide the appropriate **MFC™** defined functionality to model and visualization environments. This is demonstrated by the position of the ColorPlot_Class/ArrowPlot_Class and the ElectrostaticModel_Class. Since each of these environments is to be implemented as a DLL they require functionality from both the Yoke Class as well as the CWinApp class. Each of the ColorPlot_Class/ArrowPlot_Class and the ElectrostaticModel_Class represent classes defined by users/creators of the system. The functionality for visualization and for numerical modelling are implemented in this class through the application of classes in the Model Class Library, the Visualization Class Library and toolkits such as **VTK**.

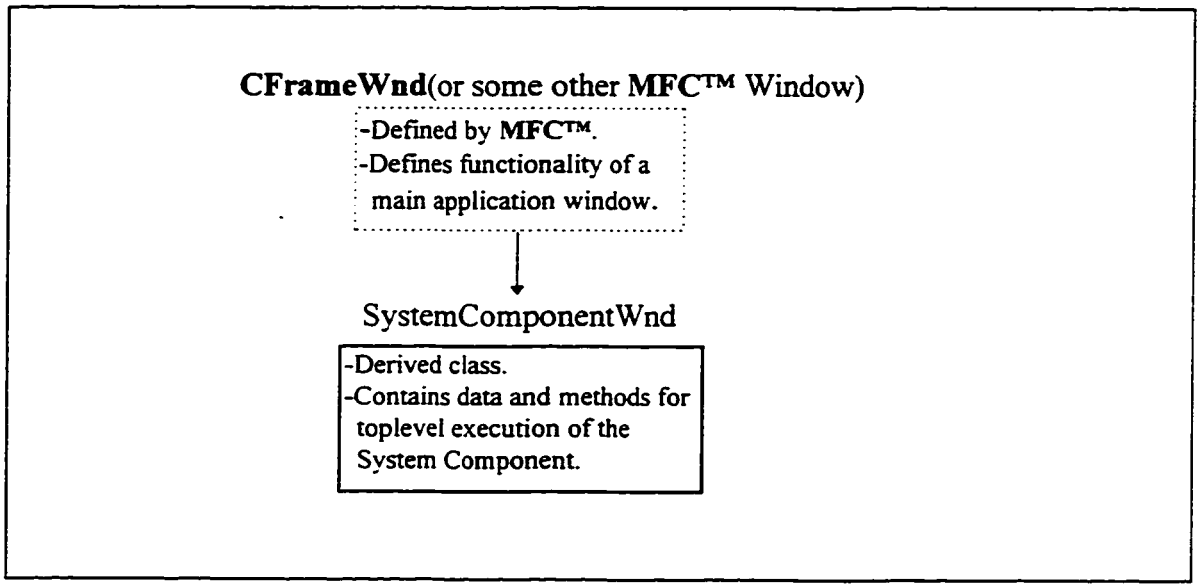

Figure B2: The SystemComponentWnd is similar to the CMainWnd of Figure A1. This class is derived from an MFC™ defined window class which manages window specific details. This window embodies the main application window which the user first interacts with once the modelling system is executed. This window provides access to modelling and visualization environments.

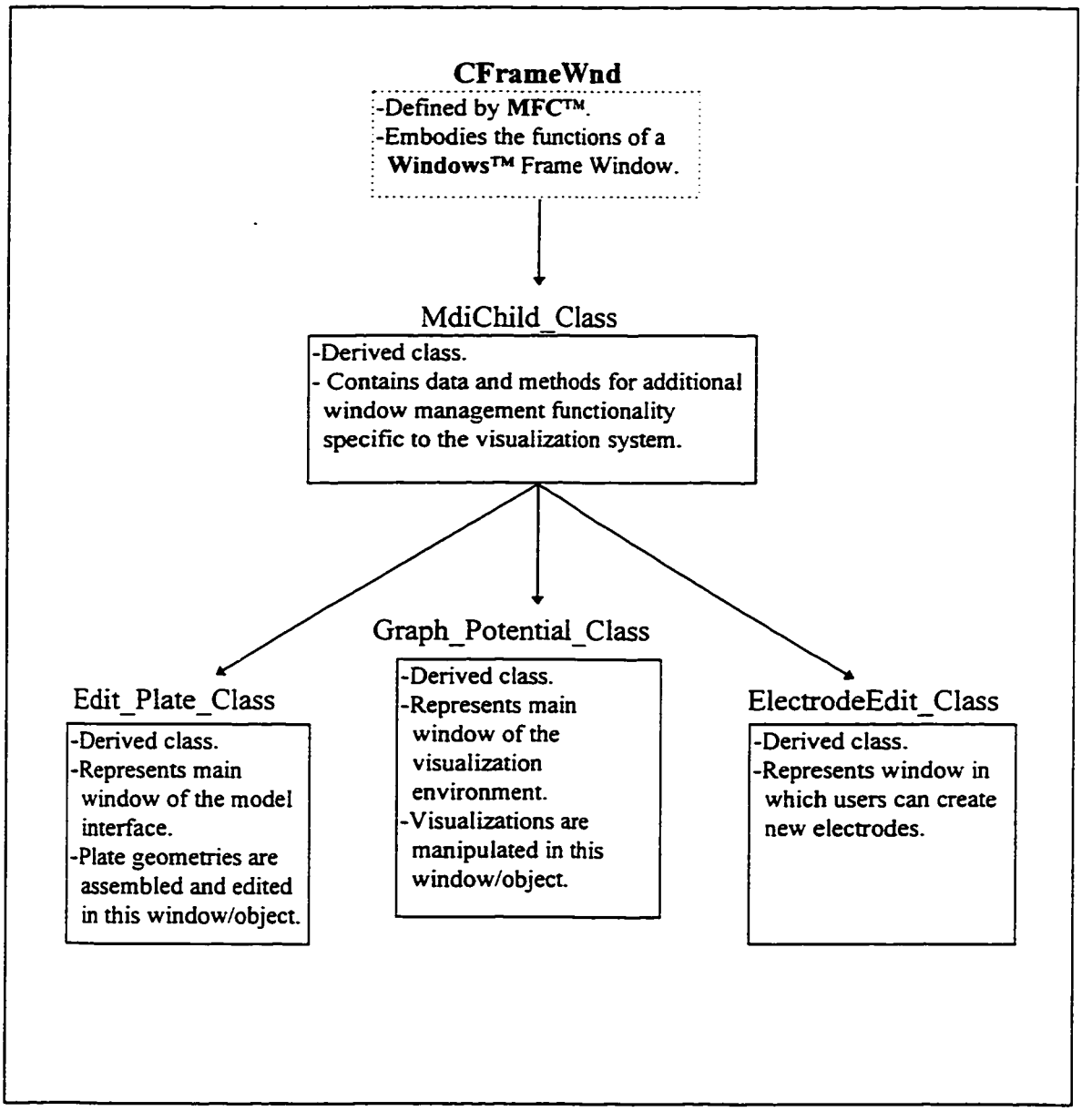

Figure B3: In this class hierarchy the bottom three classes, Edit_Plate_Class, Graph_Potential_Class and Electrode_Edit_Class, are window classes. They are derived from the MdiChild_Class which abstracts any application specific functionality that is required by the three windows. This class is derived from an MFC™ class which abstracts the window handling functionality. The Edit_Plate_Class represents the window in which the plate geometry is constructed. The Graph_Potential_Class represents the window in which a visualization is displayed, i.e. a color plot. The ElectrodeEdit_Class represents the window in which electrodes can be edited to create new ones.

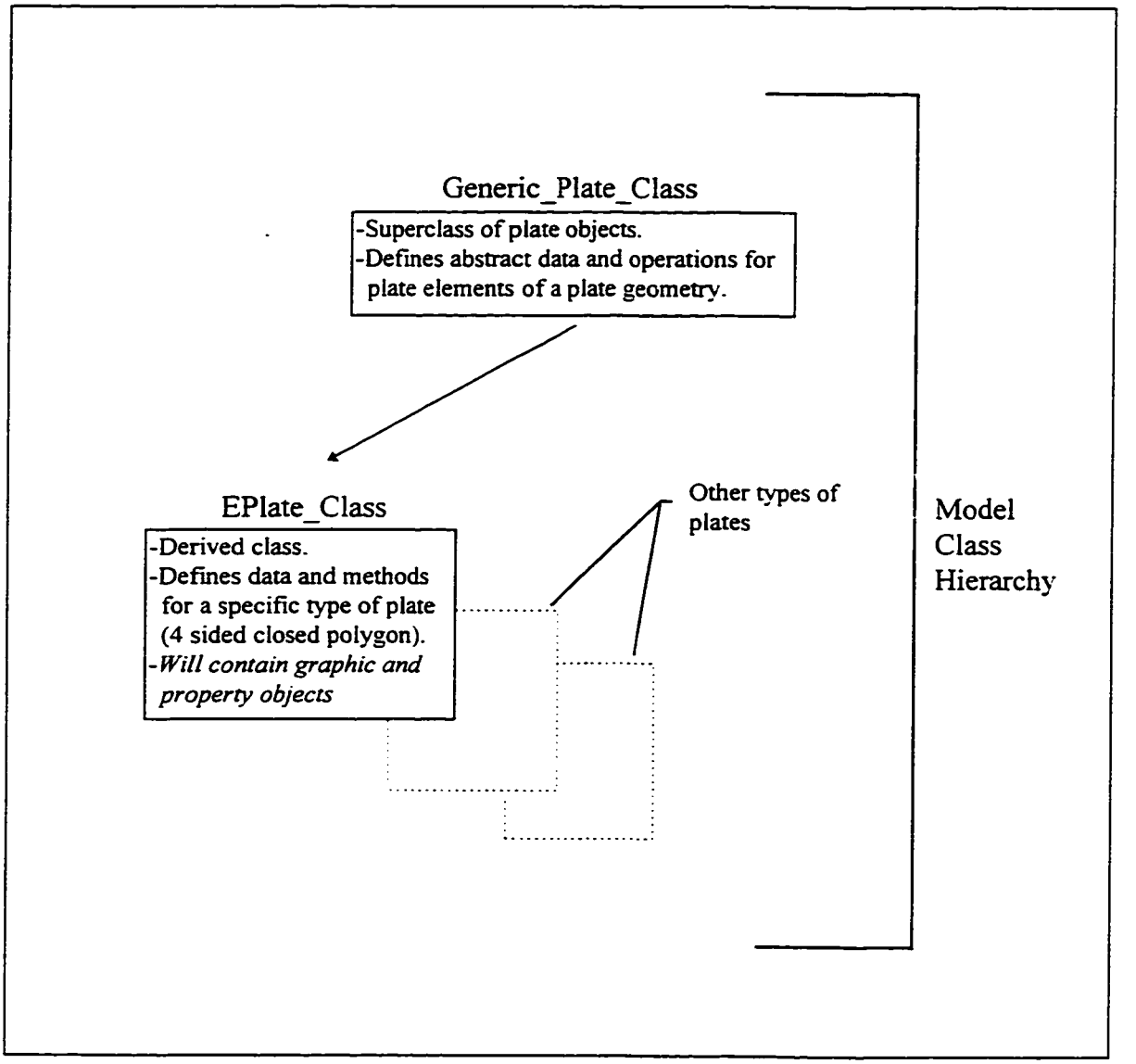

Figure B4: This class hierarchy is similar to that of Figure A3. The EPlate_Class is implemented in the same manner. The ColorPlot_Class, ArrowPlot_Class and PotentialPlate_Class have been removed. This hierarchy is a portion of the Model Class Library used to define plate elements.

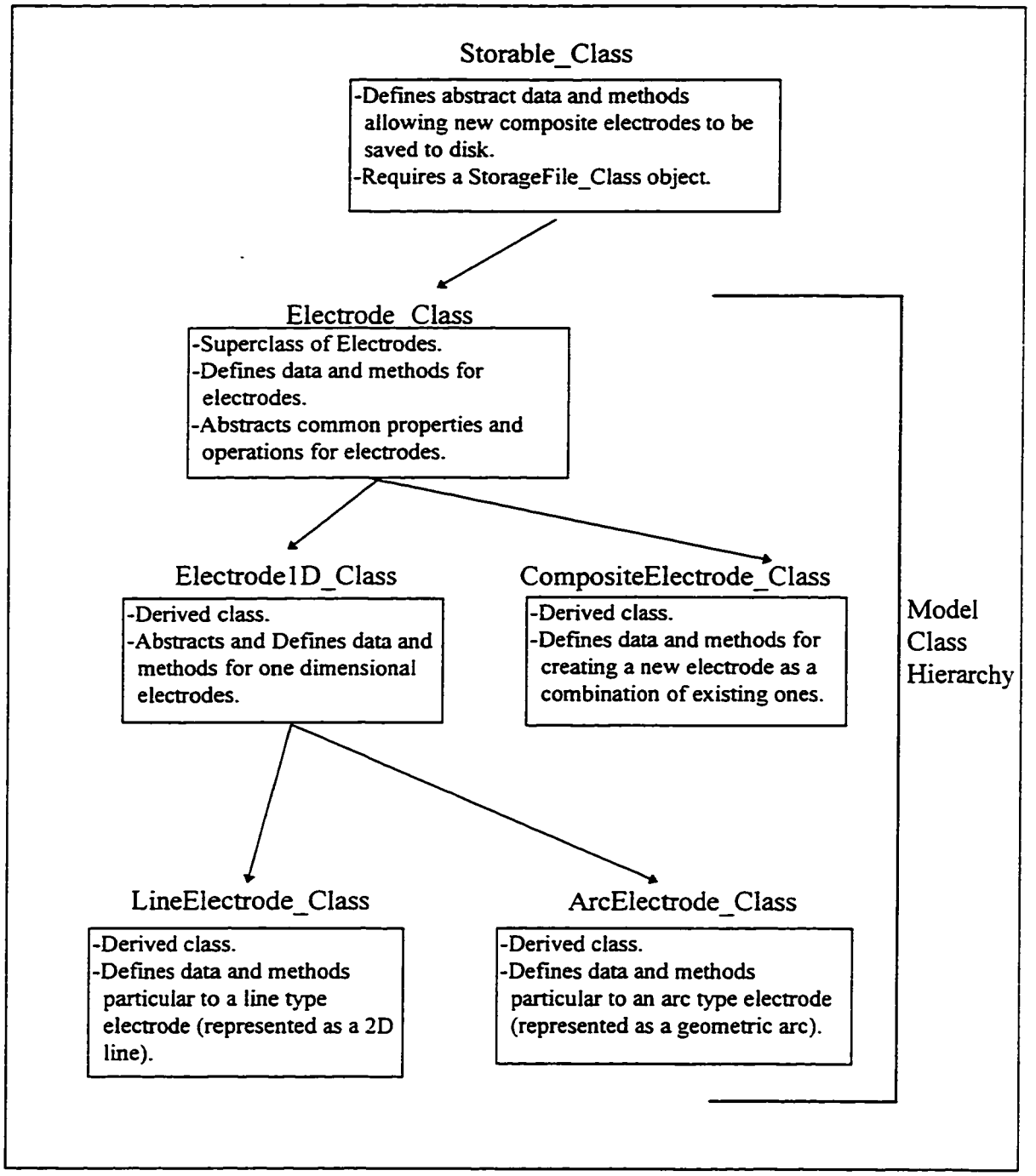

Figure B5: This class hierarchy is the same as that of Figure A4. The main difference is the Storable_Class has been moved so that individual electrodes inherit its functionality.

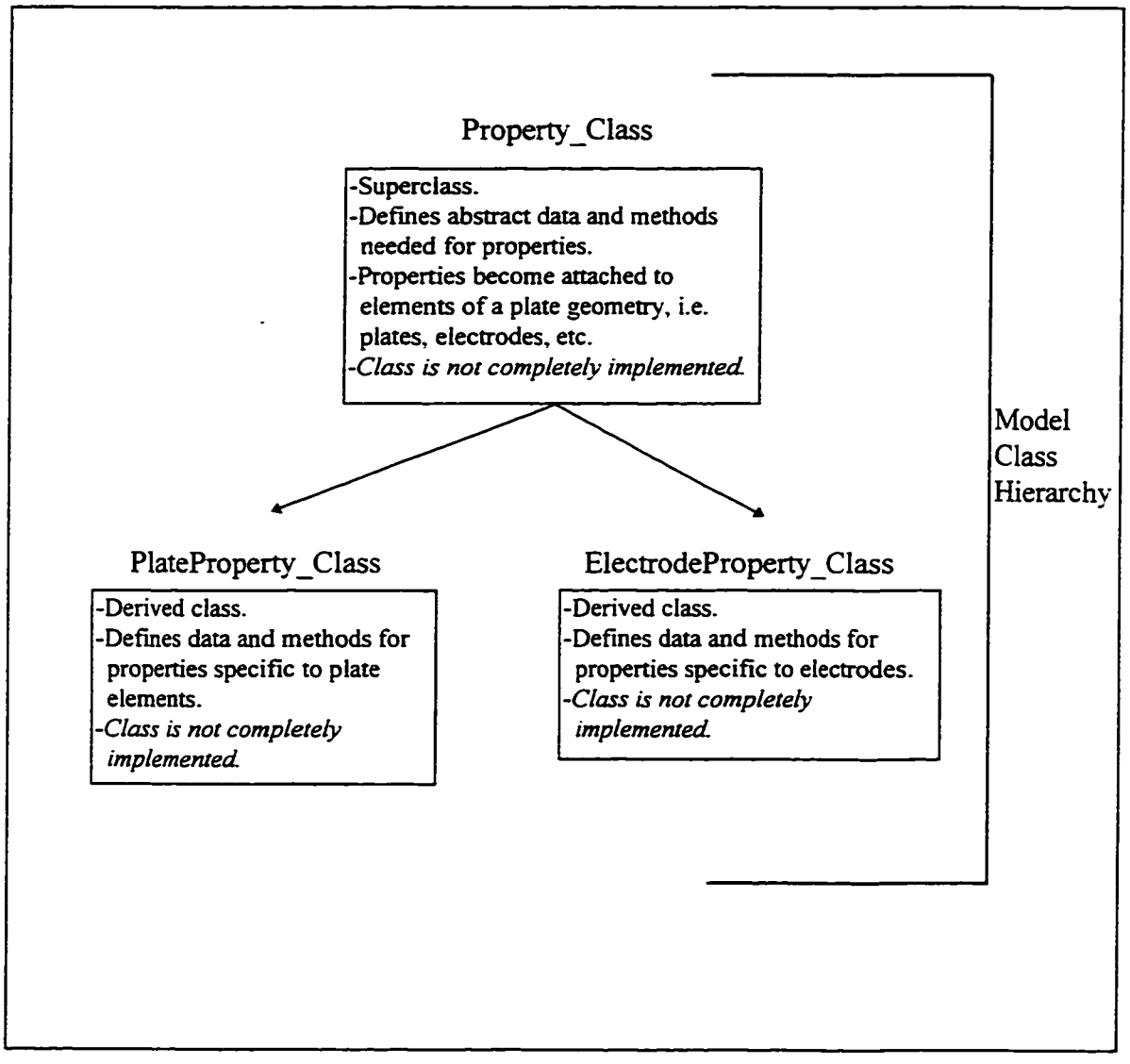

Figure B6: This class hierarchy is the same as that of Figure A5. It defines the class hierarchy of the properties used for plate geometries.

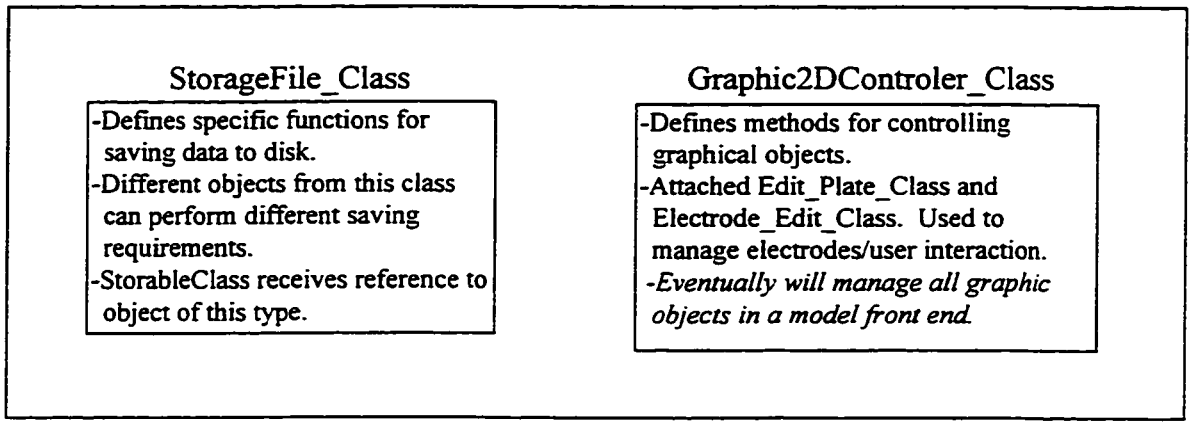

Figure B7: These two classes are the same as those of Figure A6.

Appendix C

This appendix contains a list of all the C++ source code and header files which constitute the implementation. The actual code may be obtained from the following website: *http://www.cs.uwindsor.ca/meta-index/people/csfac/rkent/welcome.html*

File: *arcelectrode_class.h/arcelectrode_class.cpp*

Description: These two files contain the definition and implementation for the class ArcElectrode_Class. This class defines and manages the functionality of this type of electrode such as its drawing and editing. This class is subclassed from the Electrode1D_Class. The ArcElectrode_Class represents one type of electrode which is used in the construction of a plate geometry.

File: *arrowplot_class.h/arrowplot_class.cpp*

Description: These files contain the definition and implementation for the class ArrowPlot_Class. This class is a subclass of Potential_Plate_Class and is responsible for creating an arrow plot visualization. Essentially this class represents a visualization environment. In this implementation this class is attached to the Graph_Potential_Class object/window in which it will display the visualization..

File: *cmainwnd_class.h*

Description: This file contains the definition of the class CMainWnd. CMainWnd class embodies the main application window which the user interacts with when the application is started.

File: *colorplot_class.h/colorplot_class.cpp*

Description: These files contain the definition and the implementation of the class ColorPlot_Class. This class is a subclass of Potential_Plate_Class responsible for creating a color plot visualization. Essentially this class represents a visualization environment. In this implementation this class is attached to the Graph_Potential_Class object/window in which it will display the visualization.

File: *compositeelectrode_class.h/compositeelectrode_class.cpp*

Description: These files contain the definition and the implementation for the class CompositeElectrode_Class. This class is used in the creation of composite Electrodes. It manages a list of individual electrodes, such as an arc or line electrode, which together constitute a new electrode with its own properties and a combination of those of the individual electrodes.

File: *defines.h*

Description: This header file contains define statements and macros.

File: *esocket_class.h/esocket_class.cpp*

Description: These files contain the definition and implementation for the socket class ESocket_Class. This class is used for accepting connections from remote machines to support distributed modelling as described in the thesis. Currently it uses the **MFC™** CSocket class.

File: *generic_palette_class.h/generic_palette_class.cpp*

Description: These files contain the definition and the implementation for the class Generic_Palette_Class. It is used for creating palettes for use with the GDI in **MFC™**.

File: *Generic_Plate_Class.h/Generic_Plate_Class.cpp*

Description: These files hold the functions and the definitions for the class Generic_Plate_Class. It is the superclass of the EPlate_Class and defines the generic functionality of plate objects which are used in the construction of a plate geometry. These files also contain the definition and implementation of the EPlate_Class. The .h file also contains the definitions for the EVertex class and the EFloat_Point class.

File: *geometric.h*

Description: This file contains geometric classes such as the EPoint3D class, the EVector3D class, the EVector2D, the EPoint2D, and the EFace class.

File: *Graph_Potential_Class.h/Graph_Potential_Class.cpp*

Description: These files contain the functions and the definitions belonging to the Graph_Potential_Class. This class is used to provide a window to which color plot objects or arrow plot objects can be attached. The ColorPlot_Class and ArrowPlot_Class draw to the window provided by the Graph_Potential_Class. This class is a subclass of the MdiChild_Class.

File: *graphic2dcontroler_class.h/graphic2dcontroler_class.cpp*

Description: These files contain the definitions and the implementation of the class Graphic2DControler_Class. It is a class that will handle the drawing and updating of all 2D graphics on the plate geometry and allow the user to manipulate any of the Electrodes on the plate. At this point it is mainly used to handle the electrodes which are inserted into a plate geometry as well as when new electrodes are being created in the ElectrodeEdit_Class.

File: *electrode_class.h/electrode_class.cpp*

Description: These files contain the definitions and the implementation for the class Electrode_Class. It is the superclass for all graphical objects used as electrodes which are inserted into the plate geometry.

File: *electrode1d_class.h/electrode1d_class.cpp*

Description: These files contain the definitions and the implementation for the class Electrode1D_Class. It is the base class for all 1D Electrodes and subclass of Electrode_Class. It is used to handle specific implementation details of 1D electrodes.

File: *electrodeedit_class.h/electrodeedit_class.cpp*

Description: These files contain the definitions and the implementation for the class ElectrodeEdit_Class. This class is a subclass of the CMdiChild_Class and provides an environment in which users can create composite electrodes.

File: *electrodeproperty_class.h*

Description: This file contains the header information and class definition for the class ElectrodeProperty_Class. It is a class which contains the more application specific property information for electrode elements.

File: *lineelectrode_class.h/lineelectrode_class.cpp*

Description: These files contain the definition and the implementation for the class LineElectrode_Class. It is a subclass of the Electrode_Class and it defines the behaviors and the functionality for manipulating a line electrode.

File: *mdi_superclass.h/mdi_superclass.cpp*

Description: The .h file contains the class description of the MdiChild_Class class. it is a subclass of CMDIChildWnd class and is the super class of the Edit_Plate_Class, the Graph_Potential_Class and the ElectrodeEdit_Class. The .cpp file contains functionality for accessing window information (number of active windows) which is inherited into the subclasses.

File: *mdichild_classes.h/mdichild_classes.cpp*

Description: These files contain the definition and functions for the Edit_Plate_Class. This class represents the main window in which the plate geometry is created. The .cpp file also contains the implementation for the EPlate_Class, the EFloat_Point class, the EPoint3D class, the Evertex class, the EPoint2D class and the EVector3D class

File: *mditest.cpp*

Description: This file contains the implementation of the main application object and application window object. This file and object are representative of the System Component.

File: *plateproperty_class.h*

Description: This file contains the header information and class definition for the class PlateProperty_Class. It is a class which contains the more application specific property information for plate elements.

File: *potential_plate_class.h/potential_plate_class.cpp*

Description: These files contain the definition and the implementation for class Potential_Plate_Class. It is derived from the base class Generic_Plate_Class. It defines data and methods for generating the plate geometry in the color plot or arrow plot classes for contextualizing the visualization.

File: *property_class.h*

Description: This file contains the header information and class definition for the class Property_Class. It is a class which contains abstract property information for user interface elements, i.e. plates and electrodes. It is the superclass for the ElectrodeProperty_Class and the PlateProperty_Class.

File: *record_class.h/record_class.cpp*

Definition: These files contain the definition and implementation functions for the Record_Class object. This object is used for keeping and manipulating

records regarding the active windows in the application. For instance the total number of windows and how many of them are of a specific type of window, i.e. Edit_Plate_Class window.

File: *storable_class.h*

Description: This file contains the definition for functions in the class Storable_Class. These functions define a standard interface for disk operations. Other classes inherit these functions which enable them to save themselves to disk. This class works in conjunction with the StorageFile_Class.

File: *storagefile_class.h/storagefile_class.cpp*

Description: These files contain the definition and implementation for the class StorageFile_Class. It is a class which must be defined in any class which want to save itself to disk. This class encapsulates the file operations to save and retrieve from disk and works in conjunction with the Storable_Class.

File: *view.h/view3d.cpp*

Description: These files contain the definition and implementation for class View_3d. This class provides functionality for 3D viewing manipulations for objects such as a plate geometry.

BIBLOGRAPHY

[Ball 96] Thomas Ball, Stephen G. Eick. "Software Visualization in the Large." *Computer*. 29(4):33-42, April, 1996

[Bancroft 90] Gordon V. Bancroft, Fergus J. Merrit, Todd C. Plessel, Paul G. Kelaita, R. Kevin McCabe, Al Globus. "FAST: A Multi-Processed Environment for Visualization of Computational Fluid Dynamics." *IEEE Proceedings of Visualization 1990*, pp. 14-26, IEEE Computer Society Press. Los Alamitos, Ca, 1990.

[Beshers 94] Clifford Beshers, Steven Feiner. "Automated Design of Data Visualizations." *Scientific Visualization: Advances and Challenges*, pp. 87-102, Academic Press LTD., New York, 1994.

[Booch 94] Grady Booch. "Object Oriented Analysis and Design with Applications." 2nd Ed., Benjamin/Cummings Publishing Company, Inc., New York, 1994.

[Brittain 90] Donald L. Brittain, Josh Aller, Michael Wilson, Sue-Ling C. Wang. "Design of an End-User Data Visualization System." *IEEE Proceedings of Visualization 1990*, pp. 323-328, IEEE Computer Society Press, Los Alamitos, Ca, 1990.

[Bronts 95] G.H.W.M. Bronts, S.J. Brouwer, C.L.J. Martens, H.A. Proper. "A Unifying Object Role Modelling Theory." *Information Systems*. 20(3):213-235, 1995.

[Bryson 94] Steve Bryson. "Real-Time Exploratory Scientific Visualization and Virtual Reality." *Scientific Visualization: Advances and Challenges*, pp. 65-86, Academic Press LTD., New York, 1994.

[Butler 93] David Butler, James C. Almond, R. Daniel Bergeron, Ken W. Brodlie, Robert B. Haber. "Visualization Reference Models." *IEEE Proceedings of Visualization 1993*, pp. 337-342, IEEE Computer Society Press, Los Alamitos, Ca, 1993.

[Causse 94] Sylvain Causse, Frédéric Juaneda, Michel Grave. "Partitioned Objects: Sharing for Visualization in Distributed Environments." *Scientific Visualization: Advances and Challenges*, pp. 251-264, Academic Press LTD., New York, 1994.

[Cendes 89] Zoltan J. Cendes. "Unlocking the Magic of Maxwell's equations." *IEEE Spectrum*. 26:29-33, April, 1989.

[Cendes 90] Zoltan J. Cendes. "EM simulators = CAE tools." *IEEE Spectrum.* 27:73-77, November, 1990.

[Creasy 96] P.N. Creasy, H.A. Proper. "A generic model for 3-dimensional conceptual modelling." *Data and Knowledge Engineering.* 20:119-161, 1996.

[Davis 84] M. E. Davis. "Numerical Methods and Modelling for Chemical Engineers", John Wiley & Sons, New York, 1984.

[Earnshaw 94] Philip K. Robertson, Rae A. Earnshaw, Daniel Thalmann, Michael Grave, Julian Gallop, Eric M. De Jong. "Research Issues in the Foundations of Visualization (Foundations of Visualization)." *Scientific Visualization: Advances and Challenges*, pp. 479-487, Academic Press LTD., New York, 1994.

[Encarnação 94] José Encarnação, Martin Frühauf. "Global Information Visualization - The Visualization Challenge for the 21st Century." *Scientific Visualization: Advances and Challenges*, pp. 55-64, Academic Press LTD., New York, 1994.

[Foley 94] Jim Foley, Bill Ribarsky. "Next-Generation Data Visualization Tools." *Scientific Visualization: Advances and Challenges*, pp. 103-128, Academic Press LTD., New York, 1994.

[Fracchia 95] David F. Fracchia, Loki Jörgenson, Ronald Kriz, Barbara Mones-Hattal, Bernice Rogowitz. "Is Visualization Struggling under the myth of Objectivity." *IEEE Proceedings of Visualization 1995*, pp. 412-415, IEEE Computer Society Press, Los Alamitos, Ca. 1995.

[Fruhaüf 94] Thomas Fruhaüf, Martin Göbel, Helmut Haase, Kennet Karlsson. "Design of a Flexible Monolithic Visualization System." *Scientific Visualization: Advances and Challenges*, pp. 265-286, Academic Press LTD., New York, 1994.

[Gallop 94] Julian Gallop. "Underlying Data Models and Structures for Visualization." *Scientific Visualization: Advances and Challenges*, pp. 239-250, Academic Press LTD., New York, 1994.

[Haber 90] Robert B. Haber, David A McNabb. "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems." *Visualization in Scientific Computing*, pp. 74-93, IEEE Computer Society Press, Los Alamitos, Ca, 1990.

[Hultquist 92] J.P.M. Hultquist, E.L. Raible. "SuperGlue: A Programming Environment for Scientific Visualization." *IEEE Proceedings of Visualization 1992*, pp. 243-250, IEEE Computer Society Press, Los Alamitos, Ca, 1992.

[Kochevar 93] Peter Kochevar, Zahid Ahmed, Jonathan Shade, Colin Sharp. "Bridging the Gap Between Visualization and Data Management: A Simple Visualization Management System." *IEEE Proceedings of Visualization 1993*, pp. 94-101, IEEE Computer Society Press, Los Alamitos, Ca, 1993.

[Legensky 91] Steve M. Legensky. "Advanced Visualization on Desktop Workstations." *IEEE Proceedings of Visualization 1991*, pp. 372-378, IEEE Computer Society Press, Los Alamitos, Ca, 1991.

[Lucas 92] Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, Kevin P. McAuliffe. "An Architecture for a Scientific Visualization System." *IEEE Proceedings of Visualization 1992*, pp. 107-114, IEEE Computer Society Press, Los Alamitos, Ca, 1992.

[Martincic 97] Fernando Martincic. "Forms-based Data Acquisition and Delivery Model Using Java for Internet Distribution." *Honours Bachelor of Computer Science Thesis*, University of Windsor, Windsor, 1997.

[Matthews 96] Gary Jason Mathews, Syed S. Towheed. "WWW-based data systems for interactive manipulation of science data." *Computer Networks and ISDN Systems*. 28:1857-1864, 1996.

[Max 1] Maxwell users' manual, *Getting Started: A 2D Electrostatic Problem*, Ansoft Corporation 1994, Four Station Square, Suit 660, Pittsburgh, PA 15219.

[Max 2] Maxwell users' manual, *Getting Started: A Magnetic Force Problem*, Ansoft Corporation 1996, Four Station Square, Suit 660, Pittsburgh, PA 15219.

[Max 3] Maxwell users' manual, *Getting Started: A 2D Parametric Problem*, Ansoft Corporation 1995, Four Station Square, Suit 660, Pittsburgh, PA 15219.

[MFC 95] *Microsoft Visual C++ Books On-line*, Visual C++ 4.0, 1994-1995 Microsoft Corporation.

[Pang 94] Alex Pang, Naim Alper. "Mix&Match: A Construction Kit for Visualization." *IEEE Proceedings of visualization 1994*, pp. 302-309, IEEE Computer Society Press, Los Alamitos, Ca, 1994.

[Pressman 92] Roger S. Pressman. "Software Engineering A Practitioners Approach." 3rd Ed., McGraw-Hill, Inc., New York, 1992.

[Ribarsky 92] William Ribarsky, Bob Brown, Terry Myerson, Richard Feldmann, Stuart Smith, Lloyd Treinish. "Object-Oriented, Dataflow Visualization Systems A Paradigm Shift?" *IEEE Proceedings of Visualization 1992*, pp. 384-388, IEEE Computer Society Press, Los Alamitos, Ca, 1992.

[Robertson 94] Philip Robertson, Lisa De Ferrari. "Systematic Approaches to Visualization: Is a Reference Model Needed." *Scientific Visualization: Advances and Challenges*, pp. 287-305, Academic Press LTD., New York, 1994.

[Rosenblum 94] *Scientific Visualization Advances and Challenges*, L. Rosenblum, R.A. Earnshaw, J. Encarnação, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, D. Thalmann, Eds. Academic Press LTD., New York, 1994.

[Sabella 89] Paolo Sabella., Ingrid Carlbom. "An Object-Oriented Approach to the Solid Modeling of Empirical Data." *IEEE Computer Graphics and Applications*, 9:24-35, Sept., 1989.

[Schroeder 92] W.J. Schroeder, W.E. Lorensen, G.D. Montanaro, C.R. Volpe. "VISAGE: An Object-Oriented Scientific Visualization System." *IEEE Proceedings of Visualization 1992*, pp. 219-226, IEEE Computer Society Press, Los Alamitos, Ca, 1992.

[Schroeder 96] William J. Schroeder, Kenneth M. Martin, William E. Lorensen. "The Design and Implementation Of An Object-Oriented Toolkit for 3D Graphics And Visualization." *IEEE Proceedings of Visualization 1996*, pp. 93-100, IEEE Computer Society Press, Los Alamitos, Ca, 1996.

[Silver 95] Deborah Silver. "Object-Oriented Visualization." *IEEE Computer Graphics and Applications*, 15:54-62, May, 1995.

[Treinish 92] Lloyd A. Treinish, David M. Butler, Hikmet Senay, Georges G. Grinstein, Steve T. Bryson. "Grand Challenge. Problems in Visualization Software." *IEEE Proceedings of Visualization 1992*, pp. 366-371, IEEE Computer Society Press, Los Alamitos, Ca, 1992.

[Upson 89] Craig Upson, Thomas Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, Andries van Dam. "The Application Visualization System: A Computational Environment for Scientific Visualization." *IEEE Computer Graphics and Applications*, 9:54-62, July, 1989.

[Watt 93] Alan Watt. *3D Computer Graphics*, 2nd Edition, Addison-Wesley Publishing Company, Harlow England, 1993.

[Wood 96] Jason Wood, Ken Brodlie, Helen Wright. "Visualization Over The World Wide Web And Its Application To Environmental Data." *IEEE Proceedings of Visualization 1996*, pp. 81-86, IEEE Computer Society Press, Los Alamitos, Ca, 1996.

Vita Auctoris

Eric Marcuzzi was born in a modest brick home in the South of Windsor (Ontario, Canada) on September 1, 1970. He received his secondary school education at Assumption College School and graduated in 1989. He completed his B. Sc. in Computer Science at the University of Windsor in 1994 and is currently pursuing his M. Sc. in Computer Science from the same University. He expects to complete his degree in the summer of 1997 and, together with his wife and two cats, hopes to live happily ever after.